

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

The
Pragmatic
Programmers

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

七周七 Web开发框架



[美] Jack Moffitt Fred Daoud 著

张霄翀 邱俊涛 孙镌宸 顾宇 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

The
Pragmatic
Programmers

Seven Web Frameworks in Seven Weeks

Adventures in Better Web Apps

七周七 Web开发框架



[美] Jack Moffitt Fred Daoud 著

张霄翀 邱俊涛 孙镌宸 顾宇 译

人民邮电出版社

北京

图书在版编目 (CIP) 数据

七周七Web开发框架 / (美) 墨菲特 (Moffitt, J.),
(美) 达乌德 (Daoud, F.) 著; 张霄翀等译. — 北京:
人民邮电出版社, 2015. 8
ISBN 978-7-115-38843-8

I. ①七… II. ①墨… ②达… ③张… III. ①网页制
作工具—程序设计 IV. ①TP393.092

中国版本图书馆CIP数据核字(2015)第121154号

版权声明

Simplified Chinese-language edition Copyright © 2015 by Posts & Telecom Press. All rights reserved.

Copyright © 2014 The Pragmatic Programmers, LLC. Original English language edition, entitled Seven Databases in Seven Weeks.

本书中文简体字版由 The Pragmatic Programmers, LLC 授权人民邮电出版社独家出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。

版权所有, 侵权必究。

-
- ◆ 著 [美] Jack Moffitt Fred Daoud
译 张霄翀 邱俊涛 孙镌宸 顾 宇
责任编辑 陈冀康
责任印制 张佳莹 焦志炜
- ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京鑫正大印刷有限公司印刷
- ◆ 开本: 800×1000 1/16
印张: 19
字数: 329 千字 2015 年 8 月第 1 版
印数: 1—3 000 册 2015 年 8 月北京第 1 次印刷
- 著作权合同登记号 图字: 01-2014-5440 号
-

定价: 59.00 元

读者服务热线: (010)81055410 印装质量热线: (010)81055316
反盗版热线: (010)81055315

内容提要

本书带领读者认识和学习 7 种影响现代 Web 应用并改变了 Web 开发方式的框架，以期给 Web 开发者带来启发和思考。

本书延续了同系列的畅销书《七周七语言》和《七周七数据库》的体例和风格。全书共 8 章，前 7 章介绍了 Sinatra、CanJS、AngularJS、Ring、Webmachine、Yesod 和 Immutable 共计 7 种 Web 开发框架，最后一章总结回顾了所有的知识点。书中对每一种框架的介绍，都为构建 Web 应用带来了独特而强大的思路。除此之外，书中还提供了一系列代码示例和在线资源以供参考。

本书适合有一定基础的 Web 开发人员阅读，能够帮助读者拓宽思路，激发更多的设计灵感。

序

2003 年，我带着家人乘坐去往科罗拉多州杜兰戈市的火车。在寸土寸金的年代里，狭窄的红色砂崖中狭窄的轨距曾经运转良好。而现在，火车变成了历史的遗迹，已经完全被更安全高效的汽车、飞机所取代。逝者如斯。

今天也一样，我们见证了变革。单核的计算机已死，或者正在死去。不错，作为它们的继承者，多核技术是一个奇迹。同时，它们也是一个巨大的技术挑战。我们已经熟知且依赖的编程语言不再按照预期的工作了，结果就是我们看到了新一代的编程语言正在诞生。到目前为止，还没有好心人宣布获胜者。

在这个背景下，我在 2010 年编写了《七周七语言》这本书。说实话，我没有期望会卖多少本。毕竟那只是一个在 Java 世界中讨论各种语言的书，也是一本在一切都被面向对象所控制的世界中讨论多种编程范式的书而已。但是程序员逐渐意识到了技术停滞带来的危险，并且接受学习编程语言是为了让人更聪明，从而能够更好地应对变化的理念。从这个方面来看，那本书是一个巨大的成功。

三年过去了，虽然函数式编程已经开始获得一些关注，但是仍然没有领导者。我们发现，多年之前扔进我们虚拟池中的多核芯片开始荡起一些涟漪。

仅仅将去年的窄轨道拓宽是不够的。在本书中，Fred 和 Jack 会向你展示前沿的人们如何将 Web 开发朝着它本应该的方向上发展。你会看到一个叫作 Sinatra 的传统的面向对象框架，然后你会朝着客户端进发，那里 JavaScript 正在编织传奇。你将会通过 CanJS 和 AngularJS 来学习如何实现完整的、富客户端的开发。接下来，你将进入服务器端，看看函数式编程的强大能力。你会遇到两个 Clojure 的框架：极简主义的 Ring 和以健壮著称的 Immutable。你会看到 Erlang 中基于状态机的 Webmachine。如果这还不足以颠覆你的想法，你会发现强大到不可思议的 Haskell 框架 Yesod。

“七周七某某”这套书就是用来拓展你的思维的。我非常自豪地为你带来这个系

列的下一本：《七周七 Web 开发框架》。衷心地希望本书能带你超越任何阻碍。

谨致问候

Bruce Tate

首席技术官, icanmakeitbetter.com

作者访谈

Q: 你们为什么要写这本书?

Jack: 解决问题的方法有很多种, 而程序员不断地开发出新技术, 构建出新语言以找到更好的解决方案。在编程的不同方面存在众多不同的想法, 多年来, 在感受过多次大开眼界的震撼之后, 我很想与人分享自己的经历。

构建一个 Web 应用的方式不止一种, 甚至连我自己经手过的项目都没有哪两个用的是同样的方式开发的。Web 编程拥有数百种可供选择的框架和库, 而传统的 GUI 编程却只有屈指可数的几种选择。如此多的可能性, 却没有哪个是完美的, 所以我想要探索那些最有意思的思路和技术, 然后分享给更多的读者。

Fred: 不论是通常所说的编程还是具体的 Web 开发都在以惊人的速度进化。我觉得本书是一个绝佳的机会来打破主流, 探索新的想法并发现 Web 开发中的不同方式。

选择这些框架的目的并不是想比较它们, 也不是为了让你在下个项目中选择它们。这本书更像是一次对 Web 开发中各种不同特性的框架的探索。

Q: 为什么选了这 7 个框架呢?

Jack: 我们希望能挑选出那些有着独特而强大思想的框架, 而不仅仅是那些已经备受关注的框架。我们在本书中探索了极简主义、可组合、静态类型、状态机和声明式语法等。

在某些情况下, 我们选择一些框架是源于我们想要探索的一些想法, 而另一些则只是同类框架中最简明的例子。而且我们选择了一些互相重叠最少的框架, 以确保覆盖尽可能多的思路。

Fred: 在受够了那些过于复杂的所谓“企业级”框架之后, 探索学习那些可以通

过恰如其分的代码来进行编码的框架可谓是令人心旷神怡。

我们还设想引入客户端的框架，因为客户端的 JavaScript 已经不再是代码碎片的大杂烩了，这些框架给了我们编写模块化和组织良好代码的机会。

Q：你们还想引入哪些框架？

Jack：我觉得 Play 这个框架和 Yesod 旗鼓相当，它使用 Scala 来支持静态类型，并且还有一些其他特性。我还想将基于 Elixir 的框架加进来，但是目前还没有任何一个成熟的框架，因为 Elixir 本身也还在不停的修改中。

在另一个平行宇宙中，整本书可能都是关于前端框架的，仅仅前端就足以带来足够有意思的思想。一方面，一些非常优秀且独特的 ClojureScript 框架正在不断涌现，比如 Webfui 和 Om；另一方面，Meteor 和 Derby 也在实时和协作应用中大放异彩。

Fred：Node.js 是一个本应该被包含进来的有趣的服务器端框架，但是我们更想引入一些客户端框架。当然，我们可以写一本完全使用 JavaScript 的书，但这并不是我们的目标。

Q：你们开始写这本书之后，有没有出现一些有趣的新框架呢？

Jack：我的 RSS 阅读器里存的全是关于各种新框架的文章，而且每天都会有更新。目前我打算研究的几个包括 Revel（一个使用 Go 编写的框架）和 Om（一个构建在 core.async 之上的 ClojureScript 框架）。虽然 Elixir 是一个很新的语言，但是基于它的 Web 框架已经开始不断涌现，比如 Sugar 和 Dynamo，这些我也都要尝试一下。

Fred：新的框架不断涌现，我们都快跟不上了。由于 Clojure 是我最喜爱的编程语言，我想要探索的两个框架分别为 total.js 和 Pedestal。

Q：学习这些都需要什么？

你需要 Windows、MacOSX 或者 Linux，还有你最喜欢的浏览器。每一章都会告诉你你要下载哪些工具以及对应的编程语言版本。

前言

一般而言，在开发 Web 应用时，往往在一段时间之后我们会想要用另外一种方式来实现，或者使用更好的工具来实现。虽然没有完美的框架，但是探索其他框架时展开的思路，不仅可以满足应用本身的需求，而且还可以在极大程度上帮助我们使用现有工具以不同的方式来解决问題。

这书记录了一些我们在研究开发应用程序的新思路时的一些探索。我们希望你从这个现代，而又没有经过太多探索的 Web 编程世界中找到乐趣。

为什么是 7 个框架

你很可能已经有一个用来完成手头工作的框架，或者已经习惯于某个框架。你可能喜欢它，也可能讨厌它，但是你还想知道世界上有没有更好的框架。即使你并不想换一个框架或者学习一门新语言，我们认为探索一下其他开发者的好想法也会积极地影响你的工作和思维方式。

我们是对新想法和编程充满热情的终生学习者。在有如此众多的 Web 框架和编程语言的今天，非常容易就可以获得很多快乐，学习有趣而新鲜的事物，你不大可能会觉得无聊。我们在自己的职业生涯中经历了许多框架，其中一些变成了我们的新宠，另外一些则给了我们启发，还有少数则给我们本来就熟悉的领域带来了新的思路。

这本书想要带给你对 7 种截然不同的 Web 框架的感性认识，既为你展现这些框架的关键思路，又引起你的好奇心和探索精神。我们探索的每个框架都具有某些独一无二的特质。与主流框架相比，它们是更少有人走的路，而且这些路上充满了欢乐和惊喜、探索和激励。

关于本书

这本书是 Pragmatic Bookshelf 丛书的“七周七某某”系列中的一本，这个系列还包括《七周七语言》、《七周七数据库》。本书中的每一章都讨论了一个不同的 Web 框架，而且大部分都是不同的编程语言，目的是为你提供一些开发现代 Web 应用程序的全新理念、风格和技术。

每一章都是独立的，并且都被组织成了 3 天的形式。其中我们会介绍框架，展现其独一无二的特性。章节之间并没有特别的顺序，阅读时也不需要按照特定的顺序，可以直接跳到自己感兴趣的章节开始阅读。

每个框架都因为某些独一无二的特质而被选中，当然这与它们的流行程度并没有太大关系。你可能会发现一些不论是语言本身还是框架你都没听过，但是有时候这正是那些最好的想法的藏身之所。

我们从第 1 章开始，介绍了 Ruby 世界中最为简单的框架之一 Sinatra。我们在探索这个小巧而精致的框架的过程中，会创建并测试一个书签应用。

在第 2 章，我们会学习 Web 应用的新趋势：使用基于 JavaScript 的客户端框架 CanJS，并使用 Sinatra 作为后端。我们重新实现一个书签应用，并在过程中展示可以被观察和响应的动态模型。

在第 3 章，我们介绍 AngularJS。这是另外一个客户端的 JavaScript 框架，但是使用了一种完全不同的风格。AngularJS 是声明式的，并且可以与你的 HTML 集成在一起。你只需要告诉它你想要什么，而不是如何完成。

Lisp 程序员常常会说“代码即数据”，我们在第 4 章介绍 Ring 时，你会看到 Web 应用程序也是数据。Ring 应用运行在一个既复杂又简单的抽象层上，并从函数式编程中受益。

你之前对 Web 应用程序如何工作的理解将会在第 5 章受到挑战，这一章介绍了 Webmachine，这个基于 Erlang 的框架将 HTTP 建模为一个状态机。这可以让你充分利用到这个协议的优势，而这一点恰好是其他框架所欠缺的。

第 6 章，介绍了 Yesod。通过 Haskell 的强类型、静态类型系统，避免了很多 Web

应用程序错误。如果你有损坏的连接或者没有很好地处理用户生成的内容，那么编译就会失败。

最后即第 7 章，介绍了 Immutant 框架，通过用 Clojure 包装 JBoss 系统，以及删除一些不必要的组件来重塑企业级 Java Web 框架。其结果就是产生一个你会欣然使用的企业级功能的组合。

这本书不是什么

在一本书中引入如此多的想法很难做到公正，于是我们不得不缩减一些你可能想要看到的功能，那些功能需要专门介绍该语言或者框架的书来覆盖。

不是一本 Web 编程教程

我们假设你已经对 Web 应用开发比较熟悉了，我们没有解释 HTML、CSS 以及一些 Web 应用程序的基础知识。你可能之前已经做过一两个 Web 应用了，如果没有，也不用担心，我们假设的 Web 知识其实都非常基础。

不是一本编程语言教程

我们用 5 个不同的编程语言介绍了 7 种不同的 Web 框架。其中有些语言你可能已经很熟悉了，比如 Ruby、JavaScript，另外一些则非常奇怪。限于篇幅，本书中并没有引入语言的简介，但是我们也考虑到了第一次接触这些语言的读者。即使你之前没有接触过这些语言，也应该很容易获取框架想要表达的思路。这些思路适用于任何语言。

不是一本安装或者部署手册

安装语言和框架变得越来越容易了，为了保持每一章都聚焦，我们并没有引入太多的安装或者部署的细节。大多数情况下，包管理器和构建工具会搞定这些烦琐的工作，如果你遇到了问题，请在搜索引擎中查找相关的在线文档。

示例代码及体例

我们想要在每一章中都覆盖尽可能多的内容，但是在有些情况下，我们省略了一些与正在解释的主题无关的代码，而这些代码又是正常运行应用程序所必需的。有时这些代码是由脚手架自动生成的，有时你需要自己下载代码包。每章应用里的代码都提供完整的代码包下载，请放手在下载的代码中进行修改，那样就不用什么都要从头手敲一遍了。

对于每种语言，我们都尽可能地使用了当时社区中最为流行的约定和工具。

在线资源

书中的应用程序和实例都可以在“Pragmatic Programmer”网站¹上找到。你还可以找到一个论坛和勘误表单，可以报告发现的问题，或者为将来的版本提一些建议。

我们希望你喜欢学习这7个不同的框架的探索之路，也希望这些框架中的好想法能给你更多的启发。

Jack Moffitt 和 Fred Daoud

2014年12月

¹ <http://pragprog.com/book/7web/seven-web-frameworks-in-seven-weeks>

对本书的赞誉

如果只看标题，你会以为这本书只是对不同技术进行了广度优先的分析。但是令人惊讶的是它同时在深度上的展开，对每一个不同主题的核心都做了足够的强调。如果你是一个多语言狂人，或者准备成为一个，那么这本书正是写给你的。

Jim Crossley

Immutable 团队核心成员，Red Hat 首席软件设计师

客观且清晰！不仅仅是一个介绍，更是一个好的起点。任何一个现代程序员都需要这样的广度和深度，我强烈推荐本书。

Pablo Aguiar

软件工程咨询师

这本书非常有意思，对每个框架的介绍都快速而清晰，作者不仅快速地带你学习了每个框架，而且又令人惊奇地详细品味了每个框架的主要功能：设计哲学、实现、测试以及对进一步探索的提示。本书包含两个 JavaScript 框架、一个 Ruby 框架、一个 Haskell 框架、两个 Clojure 框架和一个 Erlang 框架。如果你喜欢 Web 开发，则一定会喜欢上这本书的。

Giles Bowkett

高级开发，知名博主

我非常喜欢读这本书。事实上，Yesod 这一章给了我很多关于如何向非 Haskell 程序员介绍强类型语言的强大之处的全新想法。

Michael Snoyman

Yesod 的发明者，FP Complete 公司首席软件工程师

致 谢

我们要感谢使本书得以出版的“Pragmatic Bookshelf”团队。特别感谢我们的编辑 Jackie Carter，是她以自己的专业性和不懈的努力让本书变得更好，并支持到了最后。感谢 Bruce Tate，我们两个都是他的书的忠实读者，我们因能追随他的脚步而感到荣幸。感谢 Andy Hunt 和 Dave Thomas，他们创建了一个我们都感到非常兴奋的技术主题。

感谢下面这些技术审阅者，他们为每个框架都贡献了自己专业的建议：Konstantin Haase (Sinatra), David Luecke (CanJS), Miš ko Hevery (AngularJS), James Reeves (Ring), Justin Sheehy (Webmachine), Michael Snoyman (Yesod), Jim Crossley 和 Toby Crawley (Immutant)。此外，还有为多个章节都贡献了自己的评论和建议的审阅者：Kimberly Hagen、Kevin Wiley、Pablo Aguiar、Mick Thompson、Christopher Zorn、Nathaniel Schutta，以及 Aaron Bedra。

如果没有框架的作者们，这些如此有创新意义的框架也就不会存在，感谢 Blake Mizerany、Justin Meyer、Miš ko Hevery、Adam Abrons、Justin Sheehy、Andy Gross、Mark McGranaghan、James Reeves、Jim Crossley、Toby Crawley、Michael Snoyman，以及他们的团队和贡献者。

感谢那些在本书还在 beta 版本就帮助勘误的读者，他们使得本书变得更好。

Jack: 我要感谢我的妻子 Kim，她鼓励我编写本书，我的各种想法都先试讲给她，她还花费了很多时间来审阅本书。感谢我的两个孩子 Beatrix 和 Jasper，他们带给了我很多欢乐。感谢 Sean Johnson，他介绍了 Bruce 给我，而 Bruce 启动了整个项目。

Fred: 感谢我的妻子 Nadia，她在各个方面都给我以支持，生活中有她真好。感谢 Lily 和 Ruby，他们为这个家增添了如此多的欢乐和精彩。

很多年后，我将带着叹息在一个未知的远方向人诉说：林中曾有一条两岔路，而我选择了那条更少人走的路，它改变了所有的一切。

Robert Frost

目 录

第 1 章 Sinatra.....1	1.4.7 我们在第 3 天学到的... 34
1.1 简单的领域特定语言2	1.5 总结 35
1.2 第 1 天: 构建一个书签应用...2	1.5.1 Sinatra 的强项 35
1.2.1 你好, Sinatra.....3	1.5.2 Sinatra 的弱项 35
1.2.2 用 RSpec 来测试.....4	1.5.3 最后的思考..... 36
1.2.3 REST 风格的 API.....6	第 2 章 CanJS..... 37
1.2.4 数据持久化6	2.1 CanJS 的独一无二之处 37
1.2.5 创建和读取书签8	2.2 第 1 天: 创建对象和同步 变化 39
1.2.6 编写自动化测试9	2.2.1 你好, CanJS 40
1.2.7 更新和删除书签10	2.2.2 构建和扩展对象..... 41
1.2.8 我们在第 1 天学到的...11	2.2.3 观察属性的变化..... 44
1.3 第 2 天: 创建视图.....12	2.2.4 使用 CanJS 创建一个 前端书签应用..... 46
1.3.1 ERB 简介12	2.2.5 连接模型与服务器..... 47
1.3.2 Mustache 介绍.....17	2.2.6 渲染视图..... 48
1.3.3 Slim 介绍.....20	2.2.7 动态绑定..... 50
1.3.4 我们在第 2 天学到的...23	2.2.8 我们在第 1 天学到的... 50
1.4 第 3 天: 添加新功能24	2.3 第 2 天: 创建控制器..... 51
1.4.1 校验24	2.3.1 将控制器绑定到页面 元素上..... 52
1.4.2 块参数26	2.3.2 监听 UI 事件 53
1.4.3 过滤器27	2.3.3 使用 data()方法从页面 获取数据模型..... 54
1.4.4 为书签打上标签28	
1.4.5 添加标签的 API 支持...31	
1.4.6 使用正则表达式来匹配 路由33	

2.3.4	使用观察者实现控制器 之间的沟通	55	3.3.2	利用数据双向绑定的 优势	97
2.3.5	创建一个表单控制器	57	3.3.3	创建书签表单	98
2.3.6	我们在第 2 天学到的	60	3.3.4	关于作用域的重要注意 事项	100
2.4	第 3 天: 与模型的协作	61	3.3.5	端到端的自动化测试	102
2.4.1	添加校验	62	3.3.6	我们在第 2 天学到的	104
2.4.2	实现标签的处理	64	3.4	第 3 天: 创建过滤器和 路由	105
2.4.3	过滤书签	66	3.4.1	为书签添加标签	106
2.4.4	创建一个标签列表	69	3.4.2	构建一个标签列表	107
2.4.5	使用路由管理浏览器的 位置	71	3.4.3	通过过滤器操作数据	108
2.4.6	我们在第 3 天学到的	73	3.4.4	定义路由	112
2.4.7	对 CanJS 的创造者 Justin B. Meyer 的采访	74	3.4.5	我们在第 3 天学到的	114
2.5	总结	76	3.4.6	对 AngularJS 创建者 Miško Hevery 的采访	115
2.5.1	CanJS 的强项	76	3.5	总结	116
2.5.2	CanJS 的弱项	76	3.5.1	AngularJS 的强项	116
2.5.3	最后的思考	76	3.5.2	AngularJS 的弱项	117
第 3 章	AngularJS	77	3.5.3	最后的思考	117
3.1	概览	77	第 4 章	Ring	118
3.2	第 1 天: 使用依赖注入	79	4.1	Ring 简介	118
3.2.1	你好, AngularJS	80	4.2	第 1 天: 基础组件	120
3.2.2	创建服务	83	4.2.1	起步	121
3.2.3	换个角度来看我们的 书签应用前端	86	4.2.2	Hello, World!	121
3.2.4	使用资源服务	87	4.2.3	用 Korma 处理数据	124
3.2.5	为服务写自动化测试	89	4.2.4	用 Hiccup 把数据转化 为 HTML	130
3.2.6	我们在第 1 天学到的	93	4.2.5	使用 Compojure 处理 路由	133
3.3	第 2 天: 创建控制器和 视图	93	4.2.6	我们在第 1 天学 到的	136
3.3.1	创建控制器和使用视图 指令	95			

4.3 第2天: 拼接的模式	137	5.3.2 使用 Mustache 模板引擎	179
4.3.1 定义 API	137	5.3.3 Petite 里的模板	182
4.3.2 处理 JSON	138	5.3.4 处理多种内容类型	184
4.3.3 验证输入	140	5.3.5 我们在第2天学到的	186
4.3.4 可组合的路由	143	5.4 第3天: 照亮 HTTP 的阴暗面	186
4.3.5 我们在第2天学到的	146	5.4.1 让资源可缓存	187
4.4 第3天: 构建应用的其他方法	147	5.4.2 身份验证	193
4.4.1 Ring 中间件	147	5.4.3 我们在第3天学到的	196
4.4.2 Enlive	149	5.4.4 对 Justin Sheehy 的采访	197
4.4.3 关于测试	153	5.5 总结	198
4.4.4 我们在第3天学到的	154	5.5.1 Webmachine 的强项	198
4.4.5 对 James Reeves 的采访	155	5.5.2 Webmachine 的弱项	199
4.5 总结	156	5.5.3 最后的思考	199
4.5.1 Ring 的强项	156	第6章 Yesod	200
4.5.2 Ring 的弱项	157	6.1 Yesod 简介	201
4.5.3 最后的思考	158	6.1.1 组成部分	201
第5章 Webmachine	159	6.1.2 计划	201
5.1 Webmachine 简介	159	6.2 第1天: 你不能搞错的数据	202
5.2 第1天: HTTP 请求状态机	161	6.2.1 新手入门	202
5.2.1 起步	162	6.2.2 Hello, World	203
5.2.2 Hello, World	162	6.2.3 使用 DSL 描述数据	205
5.2.3 和资源函数一起工作	165	6.2.4 使用模型	207
5.2.4 资源函数	166	6.2.5 改变和删除模型	210
5.2.5 请求转发	169	6.2.6 我们在第1天学到的	211
5.2.6 参数化转发	170	6.3 第2天: 视图、表单和认证	212
5.2.7 我们在第1天学到的	171	6.3.1 Ye Olde 的模板语言	213
5.3 第2天: 构建应用	172	6.3.2 功能性表单	216
5.3.1 短链接	172		

6.3.3 认证和授权	219	7.3.1 消息队列	251
6.3.4 我们在第 2 天学到的	223	7.3.2 管道	255
6.4 第 3 天: 继续 Rumble	224	7.3.3 Overwatch 的管道	257
6.4.1 创建头版	225	7.3.4 我们在第 2 天学到的	262
6.4.2 创建一个发布新闻 表单	228	7.4 第 3 天: 多语言应用	263
6.4.3 查看新闻与提交评论	230	7.4.1 叠加	263
6.4.4 我们在第 3 天学到的	233	7.4.2 集群	268
6.4.5 对 Michael Snoyman 的 采访	234	7.4.3 我们在第 3 天学到的	272
6.5 总结	237	7.4.4 对 Jim Crossley 的 采访	273
6.5.1 Yesod 的强项	237	7.5 总结	275
6.5.2 Yesod 的弱项	238	7.5.1 Immutant 的强项	275
6.5.3 最后的思考	238	7.5.2 Immutant 的弱项	275
第 7 章 Immutant	239	7.5.3 最后的思考	276
7.1 Immutant 简介	239	第 8 章 结束	277
7.1.1 Immutant 的特性	240	8.1 关键想法	277
7.1.2 计划	240	8.1.1 简单性	278
7.2 第 1 天: 不仅仅是网络 基础	241	8.1.2 代码运行在何处	278
7.2.1 开始	241	8.1.3 组合	279
7.2.2 Hello, World	242	8.1.4 声明式优先于指令式	279
7.2.3 分布式缓存	244	8.1.5 动态类型和静态 类型	280
7.2.4 计划任务	249	8.1.6 状态机	280
7.2.5 我们在第 1 天学到的	250	8.1.7 交互性	281
7.3 第 2 天: 构建数据管道	251	8.2 快乐的探索吧	281

第 1 章

Sinatra

汉诺塔是一个益智游戏，里面有 3 个柱子和一些不同尺寸的盘子。而游戏规则是这样的：首先把所有的盘子都按照从小到大的顺序（最上面的盘子最小，最下面的盘子最大）堆放在第一个柱子上（盘子中间有一个孔，可以插在柱子上），目标就是将第一个柱子上的所有盘子都挪到第三个柱子上。挪动的过程中，不允许出现大盘子放在小盘子之上的情况。要解决这个问题本身其实并不困难，而具挑战性的是找出一个最简单的方案——使得完成该任务所需的步骤数最少。

Web 框架用来解决编写 Web 应用的问题。使用 Sinatra，不但可以非常容易并且轻巧地解决 Web 应用开发的问题，而且还允许你用最少的代码量来实现这一点¹。举个例子，Sinatra 版本的“Hello, World”就超乎想象的短小：

```
sinatra/hello/app.rb
require "sinatra"
get "/hello" do
  "Hello, Sinatra"
end
```

这里我们定义了请求类型（get），请求的 URI（/hello）以及请求的结果，就这么简单！这段代码的意思是：当以 HTTP 的 GET 方式来请求/hello 时，响应就是“Hello, Sinatra”。

¹ <http://sinatrarb.com>

1.1 简单的领域特定语言

Sinatra 利用 Ruby 优雅的语法为开发 Web 应用定义了一套领域特定语言。诸如 `get`, `put` 和 `post` 这样的方法名和 HTTP 的请求方法一一对应。当 HTTP 方法和被请求的 URI 匹配之后, 对应的方法体就会被执行, 最后会返回一个 HTTP 响应。领域特定语言提供了一种富有表达力, 同时又非常自然的方式来开发 Web 应用。另外, Sinatra 还特别适合开发那些为客户端提供 REST 风格 API 的服务。

Sinatra 是一个非常轻量级的框架, 几乎不需要依赖就可以运行。使用 Sinatra 可以毫不费力地开发并启动一个 Web 应用。此处我们将使用一个书签应用作为示例, 在这个应用中: 用户可以保存并查看自己的书签, 也可以标记这些书签, 并且可以根据标签进行搜索。

使用 Sinatra 来创建 REST 风格的应用程序非常容易, REST 风格的服务可以将你的应用程序发布成一组基于 HTTP 的 API。有了这些后端的 API, 你就可以很容易地使用 JavaScript 的框架来进行前端的编写工作。我们将在后边的章节里讲到这一点, 并会分别使用 CanJS 和 AngularJS 来进行演示。不过, 不一定非要使用 JavaScript 框架, 因为 Sinatra 本身就支持前端开发。

接下来我们将开始构建这个示例应用。第一天, 我们将创建书签的数据模型, 提供数据库持久化, 并且定义 REST 风格的 API。第二天, 我们将创建 HTML 视图, 并会使用各种不同的模板引擎。而第三天, 我们会添加数据模型的校验部分, 同时会加入为书签添加标记的功能, 将用到 Sinatra 的块参数、过滤器、基于正则表达式的路由匹配等特性。

1.2 第1天: 构建一个书签应用

在第一天的学习中, 我们先要配置一个可以运行的“Hello, World”程序, 以确保开发环境已经配置就绪。顺便会看一下如何编写自动化测试, 以确保代码可以正常运行。现在就正式开始编写示例, 通过这一章的学习我们将发现 Sinatra 更多的特性。

先来看看这个可以问好的 Sinatra 应用吧!

1.2.1 你好，Sinatra

首先确保你的 Ruby 和 RubyGems 已经安装好了，在命令行里输入：

```
$ ruby -v
ruby 2.0.0

$ gem -v
2.0.2
```

我们将使用 Ruby 2.0 版本，不过 Sinatra 在 Ruby 1.9 下也可以很好地工作。如果上边的这些命令不能正常工作，请访问 Ruby 的下载页¹并根据你所使用的操作系统安装相应的 Ruby 版本。如果你的系统中的 Ruby 是 1.8.7，那么请注意，虽然 Sinatra 在 Ruby 1.8.7 上也可以很好地工作，但是本书的示例代码需要在 Ruby 1.9 及以上版本中运行。

好了，现在来安装 Sinatra 本身：

```
$ gem install sinatra
```

这就是你开发 Sinatra 应用所需要的所有依赖了。我们在本章的后续部分，会在需要时安装其他的 gem。如果你想看到更详细的配置信息，请参考 Sinatra 自带的文档²。

我们来创建一个文件，命名为 app.rb：

```
sinatra/hello/app.rb
require "sinatra"
get "/hello" do
  "Hello, Sinatra"
end
```

然后运行它：

```
$ ruby app.rb
== Sinatra/1.4.3 has taken the stage on 4567
```

这里的输出告诉我们，应用程序运行在 4567 端口上了。这时打开浏览器，在地址栏输入 `http://localhost:4567/hello`，你就应该可以看到这样的一句问候语了：“Hello, Sinatra”。

¹ <http://www.ruby-lang.org/en/downloads>

² <http://www.sinatrarb.com/configuration.html>

看到这样的结果，就说明一切正常！不过，像这样手工来测试一个 Web 应用，很快就会让你非常累，而且易于出错。下面让我们来看看如何解决这个问题。

1.2.2 用 RSpec 来测试

编写测试代码来保证我们应用程序的功能是非常迫切的，因为测试过程会自动地验证应用程序是否是正常工作的。我们可以在任何时刻运行这些测试，以确保我们的修改并没有破坏应用中任何部分的功能。

基于 REST 风格 API 的 Web 应用（比如书签），非常适合进行自动化测试。由于这种应用程序只是返回纯粹的数据，而不是可视化的页面，因此测试起来会更加容易，测试本身也会非常健壮。

RSpec 是一个用来测试 Web 应用程序的自动化测试工具¹。安装了 `rspec` 和 `rack-test` 之后，我们就可以开始了：

```
$ gem install rspec rack-test
```

我们先来写一个简单的测试：以 GET 方式来请求 `/hello`，然后验证返回我们预期的值和问候语。

```
sinatra/hello/app_test.rb
require_relative "app"
require "rspec"
require "rack/test"

describe "Hello application" do
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  it "says hello" do
    get "/hello"
    last_response.should be_ok
    last_response.body.should == "Hello, Sinatra"
  end
end
```

¹ <http://rspec.info>

配置了需要被测试的应用之后，我们定义了一个 `describe` 块。在 `RSpec` 中，这种块将各个测试用例组织到不同的组中，并加上了描述信息。在 `describe` 块内，我们为每个测试用例定义了 `it` 块。`it` 块本身也有一个描述信息。这样做的原因就是，我们可以像写英语那样写测试代码：描述这个可以问好的应用程序，它会说你好。在 `it` 块中，代码会执行一些动作，并通过调用 `should` 方法来验证期望值。

使用 `rspec` 命令来运行测试：

```
$ rspec app_test.rb
```

```
Finished in 0.02436 seconds
1 example, 0 failures
```

输出中的一个点号表示一个测试已经执行成功，而底下的消息则更加详细地表明所运行过的例子没有失败。如果我们有失败的测试，比如预期“Hello, Sinatra”这个输出中有一个感叹号，那么我们会得到：

```
$ rspec app_test.rb
```

```
F
```

```
Failures:
```

```
1) Hello application says hello
```

```
Failure/Error: last_response.body.should == "Hello, Sinatra!"
```

```
expected: "Hello, Sinatra!"
```

```
got: "Hello, Sinatra" (using ==)
```

```
# ./app_test.rb:15:in `block (2 levels) in <top (required)>'
```

```
Finished in 0.01612 seconds
```

```
1 example, 1 failure
```

```
Failed examples:
```

```
rspec ./app_test.rb:12 # Hello application says hello
```

注意这次输出中没有点号了，取而代之的是一个 `F`，表示测试失败。输出中有一些其他有用的信息，包含我们在 `describe` 和 `it` 时指定的描述信息、失败发生的行号、预期值和实际值的对比等。

编写自动化测试是验证代码如预期般执行的好方法，而 `Sinatra` 和 `RSpec` 的结合使得编写自动化测试非常方便。在本章的剩余部分，我们将会不断地编写测试来验证这个书签的应用。

1.2.3 REST 风格的 API

我们现在就开始实现一个简单的书签应用。在这个应用中，用户可以保存自己的书签，可以标记这些书签，并且可以获取书签列表。这里我们不会让用户直接使用浏览器来保存书签，而是提供一个在线的应用程序，它允许用户在任何地方都能访问自己的书签。另外，它还为用户提供了一个集中的地方来存储自己的书签。

我们的服务会提供如下 REST 风格的 API：

GET	/bookmarks	- 获取一个包含了所有书签的列表
GET	/bookmarks/ID	- 按照 ID 获取一个书签的详情
POST	/bookmarks	- 创建一个新的书签
PUT	/bookmarks/ID	- 更新一个已有的书签
DELETE	/bookmarks/ID	- 删除一个书签

我们在第一个迭代中会实现对书签的增删改查（CRUD）操作。首先，我们需要完成数据的持久化部分。

1.2.4 数据持久化

我们需要一个地方来保存书签。这里我们使用 SQLite 数据库¹，至于对象关系映射则会使用 DataMapper²。这两个工具都非常小巧，并且可以很好地和 Sinatra 组合在一起。

先来安装需要的 gems：

```
$ gem install sqlite3 data_mapper dm-sqlite-adapter
```

安装之后，我们就可以使用 SQLite 和 DataMapper 了。DataMapper 会将 Ruby 的类转换成 DataMapper 中的资源，资源对应于数据库中的一张表。我们会配置 DataMapper，使其指向 SQLite 数据库，然后定义我们的数据模型。DataMapper 会负责创建表，保存数据到数据库，从数据库查找数据，并回填到数据模型中。

我们先来创建一个简单的 Bookmark 模型类：

¹ <http://www.sqlite.org>

² <http://www.datamapper.org>

```
sinatra/crud/bookmark.rb
require "data_mapper"

class Bookmark
  include DataMapper::Resource

  property :id, Serial
  property :url, String
  property :title, String
end
```

使用 `DataMapper`，创建一个资源非常直观。这里我们声明了 `Bookmark` 是一个 `DataMapper` 的资源，并且为其定义了三个属性。`DataMapper` 会将这些属性和数据库的表中的列对应起来。

在 `Sinatra` 应用中，我们需要先引入 `DataMapper` 和 `Bookmark` 类。然后使用 `DataMapper::setup` 来指定 `SQLite` 数据库，并使用 `DataMapper::auto_upgrade!` 来设置数据库中的表结构：

```
sinatra/crud/app.rb
require "sinatra"
require "data_mapper"
require_relative "bookmark"

DataMapper::setup(:default, "sqlite3://#{Dir.pwd}/bookmarks.db")
DataMapper.finalize.auto_upgrade!
```

这里配置了 `DataMapper` 创建一个 `SQLite` 的数据库，这个数据库是当前目录下的 `bookmarks.db` 文件。

对 `auto_upgrade!` 的调用会创建数据库中的表，不过如果表已经存在的话，它就什么也不做。它还会根据模型类的变化而修改表结果，比如我们在模型中加入了新的属性，它会自动更新表结构。这样每次重启之后，之前的数据都还是存在的。对于 `finalize`，另一个选项是 `auto_migrate!`。如果使用这个选项，每次启动应用时都会重新创建数据库结构。如果我们需要每次重启之后都有一个干净的数据库环境，比如用来测试的环境，则可以使用这个选项。

1.2.5 创建和读取书签

现在我们可以编写 Sinatra 中的方法来提供书签功能了。首先编写获取所有书签的功能：以 GET 方法来请求/bookmarks，然后以 JSON 的格式返回所有的书签。我们使用 dm-serializer 这个 gem 将 DataMapper 的资源转换成 JSON。

```
$ gem install dm-serializer
```

```
sinatra/crud/app.rb
```

```
require "dm-serializer"
```

```
def get_all_bookmarks
  Bookmark.all(:order => :title)
end
get "/bookmarks" do
  content_type :json
  get_all_bookmarks.to_json
end
```

将 HTTP 响应内容类型设置成 JSON，就可以将结果集转换成 JSON 了。而这里通过调用 DataMapper 的 all 方法来获取所有的书签信息。

虽然书签列表功能已经就绪，但是目前还没有可供返回的数据，因此我们需要先存入一些到数据库中。要创建一个书签，可以向/bookmarks 上 POST 一条数据，然后调用 DataMapper 的 create 方法：

```
sinatra/crud/app.rb
```

```
post "/bookmarks" do
  input = params.slice "url", "title"
  bookmark = Bookmark.create input
  # Created
  [201, "/bookmarks/#{bookmark['id']}"]
end
```

创建之后返回 201 状态码，以表示资源已经被创建，随之返回的是新创建的书签 ID。有了这个 ID，客户端就可以获取这个新创建的书签了。

如果你返回一个包含了两个元素的数组，Sinatra 会自动把第一个作为状态码，把第二个作为响应内容。你还可以返回一个包含三个元素的数组，Sinatra 会自动将其识别为：状态码、HTTP 响应头信息、响应内容。

当然也可以只返回一个状态码或者响应内容，如果没有状态码，Sinatra 会使用默认值 200。但是 201 更加明确，表明资源已经被创建。万维网联盟（W3C）为所有的 HTTP 方法定义了状态码，更详细的内容¹可以参考 W3C 的文档²。

根据请求信息创建书签时，应该注意到我们使用了 slice 方法来过滤出需要的值：URL 和标题。这可以避免数据模型被那些不需要的数据污染，并且也是一个安全保护，以防止恶意用户在我们不知情的情况下，将数据直接绑定到数据模型上。

slice 方法并不是内置的，但是可以很容易地加载到 Hash 类中：

```
sinatra/crud/app.rb
class Hash
  def slice(*whitelist)
    whitelist.inject({}) {|result, key| result.merge(key => self[key])}
  end
end
```

这个方法接受一个白名单作为参数，只有出现在白名单中的键值才会被加入最后的结果中。

要获取一个书签，我们需要处理 URI 上带有 ID 的请求：

```
sinatra/crud/app.rb
get "/bookmarks/:id" do
  id = params[:id]
  bookmark = Bookmark.get(id)
  content_type :json
  bookmark.to_json
end
```

现在，我们的书签应用算是有了一个很好的开端。

1.2.6 编写自动化测试

现在我们编写一个 RSpec 测试来确保创建一个书签是可以正常工作的。测试的策略是，先获取一个书签的列表，并记录列表的长度；然后创建一个书签，再次获取一次列表；最后验证新的列表的长度比之前的多一个。我们要还检验 POST 请求的响应值。

¹ <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

² <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>


```
sinatra/crud/app_test.rb
```

```
it "creates a new bookmark" do
  get "/bookmarks"
  bookmarks = JSON.parse(last_response.body)
  ❶ last_size = bookmarks.size

  post "/bookmarks",
    { :url => "http://www.test.com", :title => "Test" }

  last_response.status.should == 201
  ❷ last_response.body.should match(/\/bookmarks\/\d+/)

  get "/bookmarks"
  bookmarks = JSON.parse(last_response.body)
  ❸ expect(bookmarks.size).to eq(last_size + 1)
end
```

- ❶ 记录上一次请求中书签列表的长度。
- ❷ 创建书签之后，响应体应该包含一个新建书签的链接。
- ❸ 验证新的书签被创建了，预期当前的列表长度比之前的多了一个。

运行 `rspec app_test.rb` 能确保我们的应用可以正常地创建书签。

1.2.7 更新和删除书签

更新一个已有的书签也很简单。当得到一个 PUT 请求，并且这个请求带有一个书签的 ID 时，我们先从数据库中取出这条记录，然后更新它，最后返回状态码：

```
sinatra/crud/app.rb
```

```
put "/bookmarks/:id" do
  id = params[:id]
  bookmark = Bookmark.get(id)
  input = params.slice "url", "title"
  bookmark.update input
  204 # No Content
end
```

这里我们返回 204，表示无内容。因为这正是一个对于 PUT 请求合理的响应，表示服务器没有额外的信息需要返回给客户端。

我们再来看看测试：

```
sinatra/crud/app_test.rb
```

```
it "updates a bookmark" do
  post "/bookmarks",
    { :url => "http://www.test.com", :title => "Test" }
  bookmark_uri = last_response.body
  ① id = bookmark_uri.split("/").last

  ② put "/bookmarks/#{id}", { :title => "Success" }
  last_response.status.should == 204

  get "/bookmarks/#{id}"
  retrieved_bookmark = JSON.parse(last_response.body)
  ③ expect(retrieved_bookmark["title"]).to eq("Success")
end
```

- ① 先创建一个新的书签，并从响应中获得 ID。
- ② 使用这个新的 ID 来发送一次 PUT 请求，但是带上一个不同的标题。
- ③ 根据 ID 请求这个书签，然后判断标题确实被更新了。

我们的增删改查操作就剩下删除了。在 DataMapper 中，我们通过内置的 `destroy` 方法来删除一个对象：

```
sinatra/crud/app.rb
```

```
delete "/bookmarks/:id" do
  id = params[:id]
  bookmark = Bookmark.get(id)
  bookmark.destroy
  200 # OK
end
```

我们现在实现了基本的 REST 风格 API，它可以用来增删改查书签，并且代码也非常直观。你现在应该知道为什么 Sinatra 对于这类应用是非常好的选择了吧？

1.2.8 我们在第 1 天学到的

今天，我们从 Sinatra 的“Hello, World”例子开始，学习了如何构建 REST 风格的 API 来管理书签；学习了如何将 DataMapper 和 SQLite 与 Sinatra 集成起来，将书签存入数据库；同时，还学习了如何使用 RSpec 来完成自动化测试。第一天的学习非常高效，

通过将不同的部分组装起来，我们得到了一个清晰而强大的开发 Web 应用的方法。

第 1 天的自学

查阅

- Sinatra 的参考文档。
- DataMapper 的文档和示例。

实践

- 编写一个自动化测试来验证删除书签功能。
- 给书签类添加一个用来标识创建日期的属性。
- 在 Sinatra 应用中创建一个新的处理函数，可以返回按照创建日期排序过的书签列表。

1.3 第 2 天：创建视图

我们已经编写了一组基本的 REST 风格 API 来管理书签，但如果有用户界面就更好了。

虽然 Sinatra 在创建 REST 风格的 API 这一点上非常强大，但是它本身并不局限于此。你可以非常容易地在 Sinatra 中使用 HTML，一会就会看到。在 Sinatra 中，可供使用的 HTML 模板非常多，我们今天会使用其中的三个：ERB、Mustache 以及 Slim。

ERB 是 Sinatra 自带的，并且也非常易用。如果你倾向于在视图模板中使用很少的语法，并且不想在模板中使用逻辑，那么 Mustache 是一个很好的选择。而 Slim 则非常简洁，因为它会生成 HTML 标签。通过这三种模板引擎，我们一起来探索一下在 Sinatra 中切换模板引擎是多么简单。让我们从 ERB 开始吧！

1.3.1 ERB 简介

ERB（嵌入的 Ruby 代码）模板引擎使用 Ruby 来生成动态的内容。ERB 是 Ruby

标准库的一部分，所以 Sinatra 默认就支持它。创建 ERB 模板时，需要把 Ruby 代码和 HTML 组合起来，模板中的 Ruby 代码用`<%%>`或者`<%= %>`包含起来，前者用来执行语句，后者用来输出表达式的执行结果。

渲染一个 ERB 模板需要调用 `erb` 方法，然后将模板的名称传给这个方法：

```
sinatra/erb/app.rb
get "/" do
  @bookmarks = get_all_bookmarks
  ➤ erb :bookmark_list
end
```

Sinatra 会从当前目录下的 `views` 目录中查找模板文件，并且为模板名称自动加上 `.erb` 后缀。比如上边的例子中，实际的模板为：

`views/bookmark_list.erb`

在处理函数中，要将数据传递给模板，我们需要使用实例变量。将书签列表加载到 `@bookmarks` 变量中之后，就可以这样来展现它了：

```
sinatra/erb/views/bookmark_list.erb
<a href="/bookmark/new">Add New Bookmark</a>
<h2>List of Bookmarks (ERB)</h2>
<ul>
1  <% @bookmarks.each do |bookmark| %>
    <li>
2      <a href="/bookmarks/<%= bookmark.id %>">Edit</a>
3      <form action="/bookmarks/<%= bookmark.id %>" method="post">
        <input type="hidden" name="_method" value="delete">
        <input type="submit" value="Delete">
      </form>
      |
4      <a href="<%= bookmark.url %>"><%= h bookmark.title %></a>
      ( <a href="<%= bookmark.url %>"><%= bookmark.url %></a> )
    </li>
    <% end %>
  </ul>
```

① Ruby 代码需要放在`<%%>`中。这段代码会迭代 `@bookmarks` 中的值，执行一个块¹，因此这个块中的 HTML 片段会根据 `@bookmarks` 的长度被执行多次。

② 通过`<%= %>`，可以将 Ruby 代码的执行结果输出到页面：此处我们输出的是

¹ Ruby 中的代码片段相当于 JavaScript 中的匿名函数

书签的 ID。这允许我们为书签创建一个与其 ID 相关联的链接。

- ③ 要删除一个书签，需要提交一次 DELETE 请求，但是通过一个链接是无法做到这一点的，我们需要发送一个 POST 请求。这个请求会发送一个名字为 `_method` 的参数，参数的值为 `delete`。Sinatra 会自动将这个请求转换为一次 DELETE 请求。
- ④ 当展现一个书签的标题时，我们想要将标题先转义一下，因为用户在创建标题时，可以输入任何内容。要转义 HTML，我们此处调用了 `h` 方法。这个方法是我们自己编写的，为了保持模板的简单性，不应该被定义在模板中，而应该作为助手函数。

Sinatra 提供了一个 `helpers` 的方法来方便用户自定义一些助手函数，以便模板来使用。我们在 `h` 方法中使用代码块来转义 HTML：

```
sinatra/erb/app.rb
helpers do
  def h(text)
    Rack::Utils.escape_html(text)
  end
end
```

现在，在模板的任何地方都可以使用 `<%=h expr%>` 这样的表达式来转义 HTML 了。

处理 JSON 和 HTML 请求

我们之前添加过一个处理函数，当请求 `/` 时，返回整个书签列表。这对于 HTML 格式的客户端非常方便，比如用户从浏览器中访问我们的站点时，`/` 就是默认的路径。

当然也可以定义另外一个处理函数，当请求 `/bookmarks` 时，返回 JSON 格式的书签列表。但是如果可以重用之前的那个处理函数就更好了，我们可以通过读取 HTTP 请求头上的 `Accept` 来决定返回哪种格式的结果。

还有更好的方法，无须使用 `if` 语句，我们可以使用 Sinatra 的一个叫作 `ResponseWith` 的插件来完成这项工作。该插件包含在 `sinatra-contrib` 这个包中，首先来安装它：

```
$ gem install sinatra-contrib
```

然后使用 `response_with` 方法：

```
sinatra/erb/app.rb
require "sinatra/respond_with"

get "/bookmarks" do
  @bookmarks = get_all_bookmarks
  ➤ respond_with :bookmark_list, @bookmarks
end
```

现在，当请求头中包含了 `Accept: application/json` 时，该处理函数会返回 JSON 格式的结果；当请求头中包含了 `Accept: text/html` 时，则返回 HTML 格式的结果。第一个参数被当作模板的名字，`RespondWith` 插件会自动调用模板引擎来渲染该模板。比如，此处我们传入了 `bookmark_list` 作为参数，并且 `views` 目录中有 `bookmark_list.erb` 文件，那么它会启用 ERB 作为模板引擎。

使用片段

片段是用于组合起来形成一个完整模板的各个小的部分。当需要在多个页面间共享一些小的视图代码，或者只是将一个大的模板文件分解成小的部分时，片段会非常有用。我们会用片段来实现书签表单中的众多输入框。

在书签列表模板顶部，我们添加一个指向 `/bookmark/new` 的链接用来创建新书签，然后再添加一个可以渲染书签表单视图模板的处理函数。

```
sinatra/erb/app.rb
get "/bookmark/new" do
  erb :bookmark_form_new
end
```

接着我们创建模板本身，用来添加一条书签的表单和用来编辑一条已有书签的表单非常相似，可以先看一下添加新书签的模板：

```
sinatra/erb/views/bookmark_form_new.erb
<h2>New Bookmark</h2>
<form action="/bookmarks" method="post">
  <%= erb :bookmark_form_inputs %>
</form>
```

非常简单，这个表单会提交一个 POST 请求到 `/bookmarks` 上。它包含了另外一个模板：`bookmark_form_inputs.erb`。注意，此处我们可以像在处理函数中那样直接调用 `erb` 方法来渲染一个模板文件。把所有的输入框放到一个独立的模板文件可以很方便地在编辑页面中重用。

```
sinatra/erb/views/bookmark_form_edit.erb
<h2>Edit Bookmark</h2>
<form action="/bookmarks/<%= @bookmark.id %>" method="post">
  <input type="hidden" name="_method" value="put">
  <%= erb :bookmark_form_inputs %>
</form>
```

除了标题之外，另一处不同是这里的表单会发送 PUT 请求到 `/bookmarks/<id>` 上。不过对于表单的 `put` 方法并不存在，一个变通的方法是添加一个隐藏的输入框，名称为 `_method`，值为需要的 HTTP 方法，Sinatra 会自动识别并转换。

创建书签和更新书签表单中的输入框都是一样的，不同之处在于提交表单的方式及对应的 URL。我们可以把输入框作为一个片段：

```
sinatra/erb/views/bookmark_form_inputs.erb
<label>
  URL:
  <input type="text" name="url" value="<%= @bookmark && @bookmark.url %>">
</label>
<label>
  Title:
  <input type="text" name="title" value="<%= @bookmark && @bookmark.title %>">
</label>
<input type="submit" name="save" value="Save">
<a href="/">Cancel</a>
```

现在，我们已经在 `bookmark_form_new.erb` 和 `bookmark_form_edit.erb` 中使用片段重用技术了。

说到重用，可以通过添加一个布局页面使我们应用中的每个页面都可以重用布局中的内容。

```
sinatra/erb/views/layout.erb
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Bookmarking App</title>
    <link href="/css/bootstrap.min.css" rel="stylesheet">
    <link href="/css/app.css" rel="stylesheet">
  </head>
  <body>
    <div class="container">
      <h1>Bookmarking App</h1>
      <hr>
      <div>
        <%= yield %>
      </div>
    </div>
  </body>
</html>
```

```

    </div>
  </div>
</body>
</html>

```

当调用 `erb` 方法的时候, Sinatra 会自动找到这个布局页面, 因为我们的 `views` 目录下有一个 `layout.erb` 文件。注意此处的 `yield` 调用: 每个页面中的内容都会被放置在此处。

我们这里还引用了两个 CSS 文件。为了让 Sinatra 找到它们或者其他的静态文件 (比如 `js` 文件、图片等), 我们需要将这些资源放在 `public` 目录下, 或者通过配置 `:public_folder` 选项来指定一个目录:

```
set :public_folder, settings.root + '/my_static_file_folder'
```

下面我们接着讨论另一个模板引擎: Mustache。

1.3.2 Mustache 介绍

Mustache 是一个仅有几条简单语法规则的模板规范¹。它是一个很值得一看的 ERB 的替换者, 因为你通常都不会在模板中写太多的代码。它没有 `if-else` 语句, 也没有 `for` 循环。

要使用 Mustache, 我们需要两个 gem: `mustache` 和 `sinatra-mustache`:

```
$ gem install mustache sinatra-mustache
```

然后在 Ruby 代码中加上 `require` 语句, 这样就可以使用 Mustache 了:

```
sinatra/mustache/app.rb
require "sinatra/mustache"
```

和 ERB 非常像, 只需要调用 `mustache` 方法, 然后将模板名称作为参数传递给该方法即可。模板文件同样也放在 `views` 目录中, 但是以 `.mustache` 结尾。

```
sinatra/mustache/app.rb
get "/" do
  @bookmarks = get_all_bookmarks
  ➤ mustache :bookmark_list # renders views/bookmark_list.mustache
end
```

¹ <http://mustache.github.io>

和 ERB 类似，Mustache 也是正常的 HTML 文档，对于动态部分需要有特殊的语法。但是不像 ERB，你无法执行任意的 Ruby 代码。为了保持简单性，Mustache 被专门设计成不支持逻辑、不支持脚本。

它的语法十分简单。

`{{x}}` 输出属性 `x` 的值，`x` 的值中的 HTML 代码将被转义。

`{{{x}}}` 同样是输出属性 `x` 的值，但是不转义其中的 HTML 代码。

`{{#x}}` 和 `{{/x}}`，迭代 `x` 中的所有条目。

`{{>x}}` 将 `x` 作为模板片段进行渲染，等同于 ERB 中的 `<%= erb :x %>`，模板文件的查找路径为 `views/x.mustache`。

这些就是所有你需要的了，还可以在 Mustache 的文档里找到更多详情¹。

要在 Sinatra 里使用 Mustache，需要将数据赋值给实例变量，以便于这些变量可以在模板里被访问。在 ERB 中，我们可以使用和 Ruby 一样的语法来引用这些变量，比如 `@bookmarks`。在 Mustache 中，就不需要这个 `@` 符号了。要迭代所有的书签，需要使用 `{{#bookmarks}}` 和 `{{/bookmarks}}`。在迭代中，可以通过 `{{x}}` 来引用到书签的属性。比如用来渲染书签列表的模板就是这样的：

```
sinatra/mustache/views/bookmark_list.mustache
```

```
<a href="/bookmark/new">Add New Bookmark</a>
<h2>List of Bookmarks (Mustache)</h2>
<ul>
  {{#bookmarks}}
    <li>
      <a href="/bookmarks/{{id}}">Edit</a>
      <form action="/bookmarks/{{id}}" method="post">
        <input type="hidden" name="_method" value="delete">
        <input type="submit" value="Delete">
      </form>
      |
      <a href="{{url}}">{{title}}</a>
      ( <a href="{{url}}">{{url}}</a> )
    </li>
  {{/bookmarks}}
</ul>
```

¹ <http://mustache.github.io/mustache.5.html>

正如你所看到的，Mustache 的模板非常易读。用 Mustache 可以创建出用来重用的模板片段，就跟 ERB 中一样。下面就是书签表单输入框的模板片段：

```
sinatra/mustache/views/bookmark_form_inputs.mustache
```

```
<label>
  URL:
  <input type="text" name="url" value="{{bookmark.url}}">
</label>
<label>
  Title:
  <input type="text" name="title" value="{{bookmark.title}}">
</label>
<input type="submit" name="save" value="Save">
<a href="/">Cancel</a>
```

用来加载一个模板片段的语法是 `<{{>template_name}}`，其中 `template_name` 不需要带有扩展名 `.mustache`。比如，可以通过下面的代码在创建书签的页面中加载这个片段：

```
sinatra/mustache/views/bookmark_form_new.mustache
```

```
<h2>New Bookmark</h2>
<form action="/bookmarks" method="post">
  > {{> bookmark_form_inputs}}
</form>
```

对于布局页面，我们需要先在 `views` 目录下创建一个 `layout.mustache` 文件。通过调用 `yield` 方法，指定具体页面将在何处插入。由于我们并不需要转义任何 HTML，因此使用三个花括号的表达式 `{{{yield}}}`：

```
sinatra/mustache/views/layout.mustache
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Bookmarking App</title>
    <link href="/css/bootstrap.min.css" rel="stylesheet">
    <link href="/css/app.css" rel="stylesheet">
  </head>
  <body>
    <div class="container">
      <h1>Bookmarking App</h1>
      <hr>
      <div>
        {{{yield}}}
      </div>
    </div>
  </body>
</html>
```

对于 Mustache 就先说这么多,我们会在后续的章节 CanJS 和 Webmachine 中再次使用它。在 AngularJS 中,也有类似的语法`{{}}`。下面我们通过学习具有另一种特殊语法的模板框架 Slim 来结束这一天吧。(我们尝试了不同风格的模板,只是提供给你一些感性的认识。每个人对语法都有不同的喜好,而语法的选择种类也很多,所以你总会找到你喜欢的那一款的。)

1.3.3 Slim 介绍

Slim 的主要关注点是如何将标记语言变得更加简洁(见 <http://slim-lang.com>)。为此它在渲染过程中并没有使用常规的 HTML 加上特殊语法的形式,而是使用了一种完全不同的方式。

要使用 Slim, 需要安装对应的 gem:

```
$ gem install slim
```

然后需要在 Ruby 代码中 `require` 该 gem, 并使用模板名称作为参数来调用 `slim` 方法。Sinatra 会在 `views` 目录下查找以 `.slim` 结尾的相应的模板文件。

```
sinatra/slim/app.rb
require "slim"
get "/" do
  @bookmarks = get_all_bookmarks
  ➤ slim :bookmark_list # renders views/bookmark_list.slim
end
```

Slim 的语法与我们之前学习的 ERB 和 Mustache 的语法截然不同。在 Slim 中,我们在整个模板中都会使用它特定的语法,包括 HTML 标签本身。这种语法比常规的 HTML 标签设计得更加简洁。比如,一个 HTML 页面是这样的:

```
sinatra/slim/views/example.html
<html>
  <head>
    <link href="/css/bootstrap.min.css" rel="stylesheet" />
  </head>
  <body>
    <div class="container">
      <h1>Bookmarking App</h1>
    </div>
    <div id="footer">
```

```

    <small>Footer goes here</small>
  </div>
</body>
</html>

```

要生成上述的 HTML，对应的 Slim 模板会是这样的：

```
sinatra/slim/views/example.slim
```

```

html
  head
    link href="/css/bootstrap.min.css" rel="stylesheet"
  body
    .container
      h1 Bookmarking App
    #footer
      small Footer goes here

```

可以看到 Slim 模板比常规的 HTML 简洁多了。下面进一步讨论一下它的语法。

Slim 不使用尖括号，HTML 标签由每行的第一个标识符生成，并且无须闭括号。这是 Slim 简洁特性的一部分：使用缩进来决定标签的结构，Slim 会自动添加闭标签。标签的属性和内容写在标签名称之后，比如 `a href="http://pragprog.com" The Pragmatic Programmers`，而比当前标签的缩进更深一级的标签会被自动插入到子标签的位置。

为了进一步简化，如果没有指定标签的类型，Slim 会自动生成 `<div>` 标签。另外，因为 `id` 和 `class` 这样的属性非常通用，你可以使用 CSS 语法来模拟它。比如 `.container` 会生成 `<div class="container">`，而 `#footer` 则会生成 `<div id="footer">`。

我们来看一下如何在书签列表的应用中为 Slim 模板添加动态内容。

```
sinatra/slim/views/bookmark_list.slim
```

```

a href="/bookmark/new" Add New Bookmark
h2 List of Bookmarks (Slim)
ul
  - for bookmark in @bookmarks do
    li
      a> href="/bookmarks/#{bookmark.id}" Edit
      form> action="/bookmarks/#{bookmark.id}" method="post"
        input type="hidden" name="_method" value="delete"
        input type="submit" value="Delete"
      | |
      a< href="bookmark.url" = bookmark.title
      | (
      a< href="bookmark.url" == bookmark.url
      | )

```


我们来看一些 Slim 的语法。首先，连字符“-”用来求值 Ruby 代码，比如我们此处用它来迭代@bookmarks 列表。这时不需要再考虑关闭这个代码块，因为缩进本身就决定了块在何处结束。Slim 会帮我们负责其余的事情。

接下来，我们看一下用来编辑书签的链接，其中包含了标签的 ID。这里的语法和 Ruby 中的字符串的语法是一致的，动态内容会放入#{ }，表单上发送 DELETE 请求的 URL 也使用了这种语法。

还要注意标签后边跟着的“<”、“>”或者“<>”分别表示前置加空格、后缀加空格、前置后缀都加空格。如果只想要文本，而不想为其加上任何标签，则使用管道符(|)。

最后，使用“=”输出动态的内容。“=”输出时会转义 HTML 代码，而使用“==”会输出未转义的 HTML 代码。对于书签的标题和 URL，我们分别使用了“=”和“==”。

另外，我们会创建一个包含所有表单输入框的片段，这样就可以在创建和编辑页面重用它了。

```
sinatra/slim/views/bookmark_form_inputs.slim
```

```
label
  | URL:
  input< type="text" name="url" value="#{@bookmark && @bookmark.url}"
label
  | Title:
  input< type="text" name="title" value="#{@bookmark && @bookmark.title}"
input> type="submit" name="save" value="Save"
a href="/" Cancel
```

要渲染一个片段，需要调用= slim :bookmark_form_inputs。这和在 Sinatra 中调用 Ruby 的方法实际上是一样的。

```
sinatra/slim/views/bookmark_form_new.slim
```

```
h2 New Bookmark
form action="/bookmarks" method="post"
  == slim :bookmark_form_inputs
```

在编辑页面中重用这个片段：

```
sinatra/slim/views/bookmark_form_edit.slim
```

```
h2 Edit Bookmark
form action="/bookmarks/#{@bookmark.id}" method="post"
```

```
input type="hidden" name="_method" value="put"
input type="hidden" name="format" value="html"
== slim :bookmark_form_inputs
```

最后，Slim 还支持特定的 doctype。对于 HTML5，只需要指明 doctype html 即可。要了解 doctype 以及 Slim 语法的更多细节，可以参考 Slim 的参考文档¹。

```
sinatra/slim/views/layout.slim
```

```
doctype html
html lang="en"
  head
    meta charset="utf-8"
    title Bookmarking App
    link href="/css/bootstrap.min.css" rel="stylesheet"
    link href="/css/app.css" rel="stylesheet"
  body
    .container
      h1 Bookmarking App
      hr
      div
        == yield
```

这就是布局页面。注意此处对 yield 方法的调用，页面内容将插入到此处。

Slim 和 ERB 与 Mustache 相比有很大的不同。不过可以肯定的是，使用 Sinatra 从来都不缺 HTML 模板引擎。

1.3.4 我们在第 2 天学到的

在第 2 天里，我们学习了在 Sinatra 中使用不同的模板技术来生成 HTML 视图。这样我们就可以为 Web 应用程序编写用户界面了。

第 2 天的自学

查阅

- 额外的 ERB, Mustache 和 Slim 的教程及示例。
- Sinatra 支持的其他模板引擎。

¹ <http://rdoc.info/gems/slim/frames>

实践

- 用找出的模板引擎来重写书签应用的视图部分。
- 编写测试来验证你重写的视图。
- 验证 `response_with` 在 Slim 和 Mustache 中是否能像在 ERB 中一样，根据请求中的头信息返回对应的 HTML 或 JSON 格式的响应。

1.4 第3天：添加新功能

今天，我们要更加深入地学习 Sinatra，然后使用学到的知识来为书签应用添加一些新功能。通过加入验证机制来使得应用更加健壮，使用块参数和过滤器来重构和提升代码质量。最后我们将实现一个新的功能：为书签添加标签。

为了使得应用的输入更加健壮，我们从提供校验机制开始。

1.4.1 校验

我们已经可以创建和更新书签了，但是没有做任何校验。现在让我们来看看如何做到这一点。所有的书签都需要一个标题和一个 URL，并且 URL 的格式需要是合法的。使用 `DataMapper` 可以很容易地将这些数据限制添加在 `Bookmark` 模型上：

```
sinatra/validation/bookmark.rb
class Bookmark
  include DataMapper::Resource
  property :id, Serial
  ➤ property :url, String, :required => true, :format => :url
  ➤ property :title, String, :required => true
end
```

有了这个，当调用 `bookmark.save` 和 `bookmark.update` 的时候，如果输入是非法的，那么该方法就会返回 `false`。在我们这个场景中，需要返回 HTTP 错误码：400，表示请求非法。我们先来写一个 RSpec 测试：

```
sinatra/validation/app_test.rb
it "sends an error code for an invalid create request" do
  post "/bookmarks", {:url => "test", :title => "Test"}
  last_response.status.should == 400
end
```

此时运行测试，可以看到测试失败。这正是预期的行为，因为我们还没有写任何代码呢！这种方式被称为测试驱动开发（TDD），在编写实际的功能代码之前先编写该功能的测试代码。这种方式既可以保证你的所有功能都有对应的测试，还可以帮助你在实现之前就考虑如何实现一个功能以及需要考虑哪些情况。

我们通过发送一个 POST 请求来创建一条书签，但是由于 URL 是非法的，我们因此预期得到一个状态码为 400 的响应。现在来实现应用本身，我们需要检查 `bookmark.save` 的返回值，如果不为 `true`，那么就返回状态码 400：

```
sinatra/validation/app.rb
post "/bookmarks" do
  input = params.slice "url", "title"
  bookmark = Bookmark.new input
  ➤ if bookmark.save
    # Created
    [201, "/bookmarks/#{bookmark['id']}"]
  else
  ➤   400 # Bad Request
  end
end
```

现在我们已经完成了功能对应的代码，接着来运行一下测试以保证其可以通过。

如果一个书签没有 URL，或者没有标题，或者 URL 不合法，则都不会被创建。这已经覆盖了创建书签的过程，我们需要对更新操作提供同样的保护。即使一个书签本来是合法的，但是后续的更新请求也可能不合法：没有带标题、没有 URL 或者 URL 非法等。像 `bookmark.save` 一样，调用 `bookmark.update` 的时候也会返回一个布尔值来表示更新操作有没有成功。同样，我们使用状态码 400 来表示一个非法的请求：

```
sinatra/validation/app.rb
put "/bookmarks/:id" do
  id = params[:id]
  bookmark = Bookmark.get(id)

  if bookmark
    input = params.slice "url", "title"
    if bookmark.update input
      204 # No Content
    else
      400 # Bad Request
    end
  else
  end
end
```



```

    [404, "bookmark #{id} not found"]
  end
end

```

在更新已有的书签时，我们还需要确保待更新的书签是存在于数据库中的。我们用 `if bookmark` 来判断，如果这个语句返回 `false`，那么我们就返回状态码 404，表示未找到该记录。

这时候可以添加一个 RSpec 的测试来保证非法的 PUT 请求会得到状态码 400：

```

sinatra/validation/app_test.rb
it "sends an error code for an invalid update request" do
  get "/bookmarks"
  bookmarks = JSON.parse(last_response.body)
  id = bookmarks.first['id']

  put "/bookmarks/#{id}", {url => "Invalid"}
  last_response.status.should == 400
end

```

运行 `rspec app_test.rb`，确保所有测试都可以通过。下面我们将学习 Sinatra 的两个特性，它们可以帮助重构和提升原来的代码：块参数和过滤器。

1.4.2 块参数

当我们调用 Sinatra 的方法来处理一个请求时，比如 `get`，`put` 等会传递包含了参数的 URI，用 “:” 前缀的方式表示，比如 `/bookmarks/:id`。要获取这个 `id` 参数，我们使用 `params[:id]`。

这种方式已经很简单了，但是还有一种更简单的：块参数（之前使用的是 `params`）。

```

sinatra/validation/app.rb
get "/bookmarks/:id" do
  ➤ id = params[:id]
    bookmark = Bookmark.get(id)

```

使用块参数，代码会变成：

```

sinatra/blockparameters/app.rb
➤ get "/bookmarks/:id" do |id|
    bookmark = Bookmark.get(id)

```

使用块参数节省了一行代码。如果有多个参数的话，我们可以节省更多行。

值得一提的是，当从 `params` 中抽取参数时，键名对应于 URI 中参数的名字。但是对于块参数，变量名并不是根据相应的名称被抽取出来，而是根据参数顺序绑定的。比如，我们可以写这样的代码：

```
sinatra/blockparameters/app.rb
get "/test/:one/:two" do |creature, sound|
  "a #{creature} says #{sound}"
end
```

下面的 RSpec 测试验证了发送到 `/test/duck/quack` 的请求会获得 “a duck says quack” 这样的响应：

```
sinatra/blockparameters/app_test.rb
it "binds block parameters by order, not by name" do
  get "/test/duck/quack"
  last_response.body.should == "a duck says quack"
end
```

当然，清晰起见，使用与 URI 中的名称相同的变量名会更好。

1.4.3 过滤器

调用与 HTTP 动词相关的方法如 `get`, `post` 等时，会为每个请求设置一个处理函数。事实上我们可以设置一些在所有请求处理函数之前和之后运行的处理函数，即过滤器。过滤器是一个非常好的用于重构和重用代码的方式。

要在 Sinatra 中创建一个过滤器，需要调用 `before` 或者 `after` 方法，并且在调用的时候传入需要被拦截的 URI（如果不加 URI，那么默认会拦截所有的请求）以及需要被执行的代码块。写出来的代码，很像我们一直在写的 `get`, `post` 等的处理函数。事实上，我们还可以在 `before` 和 `after` 中使用块参数。

举个例子，所有调用 `/bookmarks/:id` 的请求都需要保证传入的 `id` 对应一个确实存在的书签，否则应该返回 404。另外，我们将这个书签示例保存到一个实例变量上以确保后续对其的操作不会再去查询数据库。这些都可以通过 `before` 过滤器来完成：

```
sinatra/filter/app.rb
before "/bookmarks/:id" do |id|
  @bookmark = Bookmark.get(id)
  if !@bookmark
    halt 404, "bookmark #{id} not found"
  end
end
```

有了这个过滤器，我们就可以来简化代码了。比如，之前的处理函数是这样的：

```
sinatra/blockparameters/app.rb
put "/bookmarks/:id" do |id|
  bookmark = Bookmark.get(id)
  if bookmark
    input = params.slice "url", "title"
    if bookmark.update input
      204 # No Content
    else
      400 # Bad Request
    end
  else
    [404, "bookmark #{id} not found"]
  end
end
```

在 before 过滤器中，我们创建了@bookmark 实例变量，校验 id，处理函数就可以关注在更新书签本身上了：

```
sinatra/filter/app.rb
put "/bookmarks/:id" do
  input = params.slice "url", "title"

  if @bookmark.update input
    204 # No Content
  else
    400 # Bad Request
  end
end
```

同样的，其他处理/bookmarks/:id 的处理函数就可以通过使用@bookmark 而得到简化：过滤器会按照 id 加载书签，而且如果找不到记录的话会返回 404 状态码。

1.4.4 为书签打上标签

书签应用看起来已经很不错了，不过还需要一个组织这些书签的方式。我们可以

为书签打标签：用户可以为书签添加任意数目的标签。有了标签，用户就可以按照它来进行浏览和查找。这样的系统会非常易于使用（你可以整理标签，还可以为一个书签打上不同的标签）。这和文件系统目录有所不同，在文件系统中，书签同时只能属于一个目录。

一个书签则可以有多多个标签，而每个标签可以关联多个书签。所以，书签和标签模型的关系是一个多对多的关联。这种情况下我们需要使用一个介于书签和标签之间的东西来表示“这个书签和这个标签是相关联的”。在数据库的术语中，这就是关联表。在应用程序中，我们称之为标记（tagging）。

在我们的模型中，一个书签可以有多个标记，一个标签也可以有多个标记。一个标记从属于一个书签和一个标签。更进一步，我们可以说一个书签有多个标签，一个标签有多个书签。因为标记在其中起着桥梁的作用，书签通过标记拥有多个标签，标签通过标记拥有多个书签。

每个标记都连接了一个书签和一个标签。因为每个书签都有多个标记，而且每个标签也都有多个标记，结果就是：书签可以有多个标签，而且不止一个书签会有同样的标签。在图 1-1 书签所属标签模型中书签 1 和书签 2 都各有两个标签，都有标签 2。

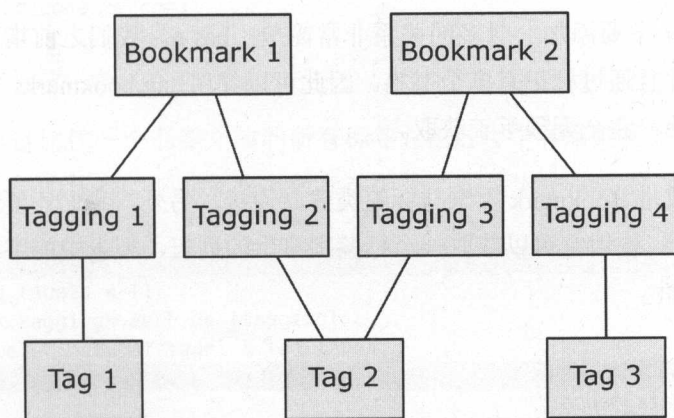


图 1-1 书签所属标签模型

好了，下面让我们来看代码吧。首先，创建 Tagging 类：

```
sinatra/tagging/tagging.rb
require "data_mapper"
```



```

class Tagging
  include DataMapper::Resource

  belongs_to :tag, :key => true
  belongs_to :bookmark, :key => true
end

```

这是一个 DataMapper 的资源。正如我们之前讨论过的，这里定义了一个 Tagging 类。因为关系表就是由键组成的，所以我们使用了 :key=>true 选项，以告诉 DataMapper 这些属性是作为主键包含进来的。

然后我们来添加 Tag 类：

```

sinatra/tagging/tag.rb
require "data_mapper"

class Tag
  include DataMapper::Resource

  property :id, Serial
  property :label, String, :required => true

  has n, :taggings
  has n, :bookmarks, :through => :taggings, :order => [:title.asc]
end

```

DataMapper 中要添加一对多的关系非常简单：has n。我们之前指明了一个标签有多个标记，并且通过标记有多个书签。因此可以使用 tag.bookmarks 来获取一个标签下的所有书签，完全无须手工获取。

我们现在改进 Bookmark 类来与标签类建立连接。另外，我们还要添加一个便利的方法：tagList。使用它可以获取一个标签字符串的列表，列表中的字符串来自标签对象的 label 属性。

```

sinatra/tagging/bookmark.rb
require "data_mapper"

class Bookmark
  include DataMapper::Resource
  property :id, Serial
  property :url, String, :required => true, :format => :url
  property :title, String, :required => true
  # Add tag support
  has n, :taggings, :constraint => :destroy
  has n, :tags, :through => :taggings, :order => [:label.asc]
end

```

```

def tagList
  tags.collect do |tag|
    tag.label
  end
end
end

```

现在，我们的模型对象已经支持标记了。

1.4.5 添加标签的 API 支持

在这个书签应用中，我们想要支持一次性添加多个标签。多个标签形成一个由逗号分隔的字符串。这个功能便于用户将任意数目的标签一次性存入，这个参数我们暂且命名为 `tagsAsString`。

为了处理 `tagsAsString` 参数，我们创建一个名为 `add_tags` 的助手函数。这个函数会按照逗号分隔字符串，并将每个子串中的空格去掉。

```

sinatra/tagging/app.rb
helpers do
  def add_tags(bookmark)
    labels = (params["tagsAsString"] || "").split(",").map(&:strip)
    # more code to come
  end
end

```

下一步通过迭代一个书签之前的所有标签，在过程中和新的标签列表做对比。我们会记录下匹配的标签，删除那些之前存在但是不在请求中的标签。

```

sinatra/tagging/app.rb
existing_labels = []
bookmark.taggings.each do |tagging|
  if labels.include? tagging.tag.label
    existing_labels.push tagging.tag.label
  else
    tagging.destroy
  end
end
end

```

最后我们会迭代整个标签列表，并创建那些不在既有标签列表中的标签。我们还需要创建一个标记来连接书签和标签，同时根据标签是否存在来决定是创建一个新的

还是复用已有的一个标签。

```
sinatra/tagging/app.rb
(labels - existing_labels).each do |label|
  tag = {:label => label}
  existing = Tag.first tag
  if !existing
    existing = Tag.create tag
  end
  Tagging.create :tag => existing, :bookmark => bookmark
end
```

当创建一个书签的时候，我们调用 `add_tags` 方法来处理打标签的动作：

```
sinatra/tagging/app.rb
post "/bookmarks" do
  input = params.slice "url", "title"
  bookmark = Bookmark.new input
  if bookmark.save
    ➤ add_tags(bookmark)

    # Created
    [201, "/bookmarks/#{bookmark['id']}"]
  else
    400 # Bad Request
  end
end
```

我们已经成功地实现了为书签添加标签的功能，并可以通过根据标签查找书签来使用。此外，我们还支持根据多个标签查找，然后返回符合所有标签的书签。搜索通过向诸如 `/bookmarks/tag1/bookmarks/tag1/tag2` 这样的 URI 发送 GET 请求而完成，多个标签之间通过一个斜杠来分割。

通过星号(*)，Sinatra 可以处理 URI 中任意数量的参数。事实上，从 URI 中获取值的参数名的方法叫作 `:splat`。因为在 URI 中可以有多个星号 (splat)，比如 `/one/*/two/*`，`params[:splat]` 会返回一个值的数组。因为我们只有一个星号，所以可以通过 `params[:splat].first` 来获取参数值。该值包含了由斜杠分割的标签，比如 `tag1/tag2`。我们可以按照斜杠来分割这个字符串，然后得到一个数组。把这些都放在一起，代码如下：

```
sinatra/tagging/app.rb
get "/bookmarks/*" do
  tags = params[:splat].first.split "/"
  bookmarks = Bookmark.all
```

```
tags.each do |tag|
  bookmarks = bookmarks.all({:taggings => {:tag => {:label => tag}}})
end
bookmarks.to_json with_tagList
end
```

获取了用于过滤书签的标签列表之后，我们便从完整的书签列表开始，然后接连使用每个标签来进行过滤，并且使用 `all` 方法加上过滤条件来逐步缩小查找范围。最后，我们以 JSON 格式来返回最终的查找结果。

你可能已经注意到了调用 `bookmarks.to_json` 的参数 `with_tagList`，让我们来看一下 `with_tagList` 的代码：

```
sinatra/tagging/app.rb
with_tagList = {:methods => [:tagList]}
```

默认的，当序列化对象时，`to_json` 不会包含关联关系，可以通过这样的方式解决：提供一个键为 `:methods`，值为一个关键字列表的哈希表。这个哈希表指明了我们想要序列化的关联关系（此处为 `:tagList`），有了它返回值中就包含了每个书签的标签列表。

1.4.6 使用正则表达式来匹配路由

我们现在有了一个用于过滤书签的路由。但是问题来了，这个路由和另外一个用以按照书签 ID 来获取书签的路由 `GET /bookmarks/<id>` 冲突了。

我们可以利用 Sinatra 的两个特点来摆脱这个两难境地：①路由可以通过正则表达式来匹配；②路由是按照它们被定义的顺序来匹配的。

当获取一个书签时，URI 中的 ID 必须是数字。我们可以使用只匹配 `bookmarks/` 之后的数字的正则表达式来限制 URI：

```
sinatra/tagging/app.rb
get %r{/bookmarks/\d+} do
  content_type :json

  @bookmark.to_json with_tagList
end
```

把按标签过滤书签的处理函数放在按 ID 获取书签的处理函数之后，我们就可以

解决这个冲突了。

我们将在 `before` 过滤器中使用相同的正则表达式，另外还有更新书签和删除书签的处理函数。

```
sinatra/tagging/app.rb
before %r{/bookmarks/(\d+)} do |id|
  # ...
end
put %r{/bookmarks/\d+} do
  # ...
end
delete %r{/bookmarks/\d+} do
  # ...
end
```

我们现在拥有一个增强版的书签 API 了。客户端在创建书签时可以带上标签，并且可以按照一个或者多个标签来过滤出一个书签的列表。

1.4.7 我们在第 3 天学到的

今天的内容丰富有趣，让我们学习到更多的 Sinatra 和 DataMapper 的特性。我们为书签添加了校验，使用了块参数和正则表达式来使得处理函数更加多样化。最后我们把剩余的时间花在实现一个新功能上，为书签打上标签。在这个过程中，我们覆盖了所有的层：数据库、模型和 Web 应用本身。

第 3 天的自学

查阅

- 创建自定义路由匹配器的文档。
- 现实中使用 Sinatra 的 Web 应用程序示例。

实践

- 使用 `curl` 发送请求来创建带有标签的书签，以及按照标签来过滤书签。
- 编写自动化测试来验证更新带有标签的书签。
- 在视图模板中添加对标签的支持。

1.5 总结

Sinatra 是一个提供了简单而又自然的 DSL 用以开发 Web 应用程序的 Ruby 框架。如果一件事可以被做得很简单，那么为什么要把它做复杂呢？使用 Sinatra 可以快速启动，并且避免那些使用大框架带来的复杂性。

Sinatra 是如此的聪明，提供了一种引人注目的方式来将已有的 Ruby 应用程序变为 Web 应用程序。正如你在本章所看到的，通过使用 RSpec 或者其他你喜爱的测试框架，Sinatra 还非常容易做单元测试。

1.5.1 Sinatra 的强项

在 Sinatra 中开发 REST 风格的 API 尤其方便，如处理路由、在 URI 中使用参数、使用正则表达式等都毫不费力，就好像返回 HTTP 状态码、消息体、JSON 格式的响应等一样轻松。

Sinatra 紧密地关注于很有限的几个功能，并把这些功能做得很好。学习这个框架不会是一个困难的任务，你会迅速变得高效又多产。

我们还看到了使用模板引擎是多么的容易。除了学习的 3 个模板，Sinatra 还支持更多的模板引擎，你可以自由地选择自己偏好的模板风格。

持久化库的使用同样方便。我们使用了 DataMapper，集成 DataMapper 和 Sinatra 毫不费力。你也可以轻松地与其他的 ORM 比如 MongoMapper，Sequel 或者 ActiveRecord 集成。

1.5.2 Sinatra 的弱项

Sinatra 的简单性和较小的适用范围是其弱项。当我们构建大型的应用程序时，需要很多 Web 开发的功能，这时就需要找到那些可以满足自己需求的库。

另外一个短处并非 Sinatra 所特有，而与它的宿主语言 Ruby 有关。Ruby 被设计得非常易于使用。Ruby 语言中的一个设计决定，就是在并行执行的复杂性和单线程

之间选择。诸如 Thin, Unicorn 和 Puma 这些服务器减轻了这方面的问题,但是毫无疑问,使用 Ruby 和其他的面向对象语言,将很可能都需要和真正多线程的应用作斗争。

1.5.3 最后的思考

Sinatra 很好地使用了 Ruby 简洁并富有表达力的语法来提供优雅而易用的 Web 框架。Sinatra 特别适合开发 REST 风格的应用,并且能无缝地与众多可以产生 HTML 页面的模板引擎集成。Sinatra 本身只关注于 Web 框架的本质,它很轻量,并且会给你使用其他库来完善应用程序技术栈的自由。

第 2 章

CanJS

宝石迷阵是这样一个游戏：有很多彩色的宝石，玩家移动相邻的宝石，让一样的宝石排成一列然后消失掉。更有趣的是，当新的宝石掉下来取代消失掉的宝石后又形成了更多的列，从而引起一系列的连锁反应。游戏高手们会事先计划好会发生哪些连锁反应，然后通过移动几个宝石来诱发之。而游戏引擎则会管理这些连锁反应。

同样的，CanJS 也是这样一个 JavaScript 框架¹。你只需要写代码去改变一处地方，它就会去处理因此而产生的一系列连锁式的更新，而不需要再去手动写代码来刷新视图、改变界面组件的状态或者向服务器发送更新请求。通过对观察者模式和动态绑定的使用，只需要改变模型，CanJS 就会自动帮你完成上述的所有事情。

2.1 CanJS 的独一无二之处

CanJS 帮助你以 MVC 的结构组织起来不能说是一个新颖的方法，因为大部分的 JavaScript 框架都遵从了 MVC 模式（或者其他非常类似的变体）。使 CanJS 脱颖而出的，是它在丰富的特性与直接易懂的方法论之间独一无二的平衡。CanJS 让你的代码可以保持模块化，并且将关注点分离。

在 CanJS 中，当对象的属性变化时，就会通知到相应的监听器。组件之间的交流

¹ <http://canjs.com>

是通过事件而不是直接使用它们的引用，这样就可以做到组件分离。CanJS 还可以通过模板加载动态视图，并且可以在模型变动时自动地更新对应的视图；同时还有一些模型可以负责管理 Ajax 请求的分发并且处理响应，用来保持客户端与服务端的一致。当然，还可以使用控制器来轻松地管理自己的应用。这些控制器可以处理用户接口事件，同时又能保证代码的模块化以及彼此之间的解耦合。另外，CanJS 还提供了路由、局部视图、数据过滤以及其他一些功能。

CanJS 在提供以上所有特性的同时，还占用了相对较小的空间。它需要以下依赖中的任意一种：jQuery¹、Zepto²、Dojo³、Moostools⁴或者 YUI⁵。在下面的例子中，我们就会使用到 jQuery。

CanJS 的内核由几个核心组件组成，下图显示了它们是怎样拼装到一起的。can.Constructor 会创建包含静态原型属性的对象，这些对象可以被继承和覆盖。CanJS 的其他大部分部件都继承自 can.Constructor，所以接下来我们会学习与理解它的工作机制，如图 2-1 所示。

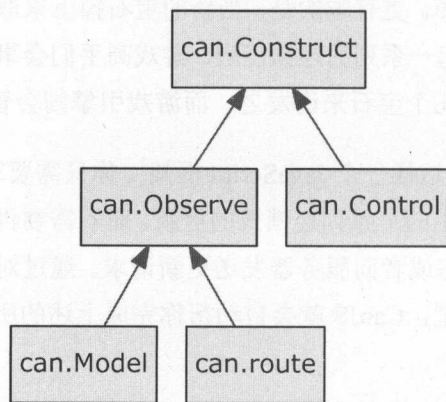


图 2-1 CanJS 核心技术栈

can.Observer 会创建一种别人可以观察的对象，当属性发生变化时就会被通知。像之前提到的那样，组件之间使用事件通信可以避免它们绑定得太紧。实际上，这种可观察的对象就是 CanJS 的精髓。

¹ <http://jquery.com>

² <http://zeptajs.com>

³ <http://dojotoolkit.org>

⁴ <http://www.mootools.net>

⁵ <http://yuilibary.com/>

`can.Module` 扩展了 `can.Observer`，使它可以同步客户端与服务端的变化。如此一来，就免去了手动分发 Ajax 请求和处理服务端返回结果的麻烦。

`can.view` 可以使用 Mustache 或 EJS¹ 加载以及渲染模板。这样不仅让你的模板简洁并易于遵循使用，还提供了动态绑定的功能。你可以只关注模型的更新，而 CanJS 会帮你实现相应视图的刷新。

`can.Control` 构建了一系列组件，将模型和视图联系在一起。我们会发现，控制器提供了一种便捷的方法去监听视图的事件；与此同时，它仍然是自己页面的一部分，不会对其他组件进行干涉。

`can.route` 通过管理浏览器的哈希实现了路由的功能，使得单页面的 CanJS 应用可以通过向后键和向前键来导航或者标记。

第一天，我们会使用到 `can.Constructor`、`can.Observer`、`can.Module` 和 `can.view`。第二天，我们会关注 `can.Control`、关联模型和视图并且处理一些界面事件。第三天，我们会深入学习模型以及怎样使用 `can.route`。

有很多有趣的特性等待我们去学习，让我们开始吧！

2.2 第1天：创建对象和同步变化

在 CanJS 之旅的第一天，我们会学习 `can.Construct` 如何通过继承创建一个层级关系。这是一个非常重要的部分，因为 CanJS 的其他部分也是基于 `can.Construct` 构建的。

接下来，我们会学习 `can.Observer`。它是 CanJS 非常有用的一个部分，负责对变化进行监听并且触发事件，以保持组件之间的独立性。我们还会看到 `can.Model` 是如何让客户端与服务端之间的同步变得更简单的。最后我们会讨论一下视图的渲染和 `can.Observer` 是如何在视图中使用动态绑定的。

首先，让我们一起来设置 CanJS 的库和依赖组件。

¹ 分别对应 <http://mustache.github.com/> 和 <http://embeddedjs.com>

2.2.1 你好, CanJS

我们要关注的第一件事情就是如何加载并使用 CanJS 和 jQuery 的最小安装集。要想在你的电脑上运行以下这种“Hello, World”类型的程序,必须确保所有东西都正常工作,而且能够修改自己的代码来做一些实验。

在 canjs/public 文件夹下, CanJS 和 jQuery 按照如下形式安装在 lib 目录下:

```
+lib
|+canjs.com-1.1.8
| `--can.jquery.js
| `--(other can.*.js files)
`+jquery
  `--jquery-1.10.2.js
```

现在,再打开 canjs/public/index-basic.html 文件。这是一个普通的 HTML 页面,通过标准的<script>标记加载了 CanJS 和 jQuery。

canjs/public/index-basic.html

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>CanJS Basic</title>
  </head>
  <body>
    <div id="result"></div>
  </body>
  > <script src="lib/jquery/jquery-1.10.2.js"></script>
  > <script src="lib/canjs.com-1.1.8/can.jquery.js"></script>
    <script src="index-basic.js"></script>
  </html>
```

最后一个文件 index-basic.js,是你真正使用 CanJS 写代码的地方。你可以选择自己最喜欢的编辑器打开这个文件,就会看到这时里面只包含了一些简单的足以加载 jQuery 和 CanJS 的代码。

canjs/public/index-basic.js

```
$(document).ready(function() {
  // Use can for CanJS
  var $result = $("#result");
```

```

can.each(["One", "Two", "Three"], function(it) {
  $result.append(it).append(", ");
});
$result.append("Go CanJS!");
});

```

在浏览器里打开 `index-basic.html`，如果你可以看到 “One, Two, Three, Go CanJS”，就说明 jQuery（以 \$ 开头的代码）和 CanJS（对 `can.each` 的调用）都已经可以工作了。就是这么简单！

恭喜，你已经准备好开始探索 CanJS 了。你可以在 `basic.html` 和 `index-baisc.html` 文件里，写一些自己的代码来试试。

现在让我们一起来探索一下 CanJS 不同的活动部件，首先是基础的构建模块：`can.Construct`。

2.2.2 构建和扩展对象

正如你在图 2-1 CanJS 核心栈中看到的那样，CanJS 的很多部分都继承自 `can.Construct`。下面我们来具体看一下。

JavaScript 是一个无类别的语言，使用了原型继承而不是典型继承。这些涉及的细枝末节，对我们来说都是一种很大的挑战。`can.Construct` 通过提供工厂方法来减小复杂性，可以让我们简单而快速地使用通用的属性构建对象。`can.Construct` 是一个非常基础的方法，用户可以调用它来构建一种单父类继承的层级关系，也可以调用父类的方法或者是在子类覆盖父类的方法。

想要使用 `can.Construct`，可以用一个包含着属性的对象调用它的拓展方法，然后就可以得到一个构造函数，接下来就可以使用 `new` 关键字和这些属性来创建对象。以下是一个例子：

```

canjs/public/concepts/concepts-test.js
var Example = can.Construct.extend({
  count: 1,
  increment: function() {
    this.count++;
  }
});

```



```
var example = new Example();
example.increment(); // example.count is now 2
```

如果想在创建一个新的对象时传入参数，需要定义一个 `init` 方法：

```
canjs/public/concepts/concepts-test.js
var Example = can.Construct.extend({
  init: function(count) {
    this.count = count;
  }
});
var example = new Example(42); // example.count is 42
```

这里有一个关于继承的重点：可以通过调用父类的构造方法（不使用 `new` 关键字）来传递子类的属性。一个继承自父类的子类，可以添加新的属性或者覆盖父类的属性。子类还可以使用 `_super` 关键字来调用父类的方法：

```
canjs/public/concepts/concepts-test.js
var Parent = can.Construct.extend({
  init: function(count) {
    this.count = count;
  },
  increase: function() {
    this.count++;
  },
  read: function(prefix) {
    return prefix + " " + String(this.count);
  }
});

var Child = Parent({
  // Child inherits the init function

  // Override increase
  increase: function() {
    this.count += 10;
  },
  // Add new function: decrease
  decrease: function() {
    this.count--;
  },
  // Override read, but call parent's version
  read: function() {
    return this._super("Count is") + "!";
  }
});
```

```

var child = new Child(2); // calls Parent's init
child.increase(); // calls Child's increase
child.decrease(); // calls Child's decrease
child.count; // returns 11
child.read(); // returns "Count is 11!"

```

最后当使用两个参数来调用 `can.Construct` 时，第一个参数定义了对象的静态属性，而第二个参数是我们一直使用至今的一个原型属性（或实例属性）。这里有一个关于它的例子：

```

canjs/public/concepts/concepts-test.js
var Example = can.Construct.extend({
  staticCount: 0,
}, {
  protoCount: 0
});

var example1 = new Example();
var example2 = new Example();

example1.constructor.staticCount = 2;
example1.protoCount = 2;

Example.staticCount; // returns 2
example2.constructor.staticCount; // returns 2
example2.protoCount; // returns 0

```

注意例子中的 `Example.staticCount`，可以看到静态属性是如何通过构造属性的一个实例获得的或者直接通过构造方法获得的。

要记住如果你只向 `can.Construct` 传一个参数，就需要自己定义原型属性。如果你需要一个只带有静态属性的构造方法，那就需要传递两个参数，并且第二个参数应该是一个空的对象，如下例所示：

```

canjs/public/concepts/concepts-test.js
var ExampleStatic = can.Construct.extend({
  staticCount: 4
}, {
});

```

学会如何定义一个含有静态属性的构造方法是非常有用的，且基本上就是创建对象的时候需要掌握的全部知识了。接下来，在我们学习 `can.Model` 的时候就可以体会到。

CanJS 中剩下的主要部分, 比如 `can Observe`, `can.Model` 或 `can.Control`, 都是基于 `can.Construct` 的。接下来就让我们一起继续 CanJS 的学习之旅: `can.Observe`。

2.2.3 观察属性的变化

CanJS 应用的灵魂和核心就是具有监听属性的对象, 或者简单来说就是“观察者”。

`Can.Observe` 提供了观察对象属性值变化的功能。这一点是非常重要的, 因此可以让应用的各组件之间完全解耦。当在构建应用的时候, 代码块都是相互独立的, 彼此之间通过观察者对象来进行交流。这样你就绝对不会写出一团乱麻般不可维护的代码。

假设有一个在线的书店, 在页面上可以看到一些用户界面组件: 书的列表, 剩余库存量, 购物车中书的总览和总价。当你把一本书加入购物车的时候, 每个组件都需要刷新来显示最新的值。正如你在图 2-2 中看到的那样, 如果让“加入购物车”组件去刷新其他的组件, 最终将不得不让组件之间都拥有彼此的直接引用。如此一来, 组件之间就互相绑定而不是相互独立, 这是非常不好的现象。

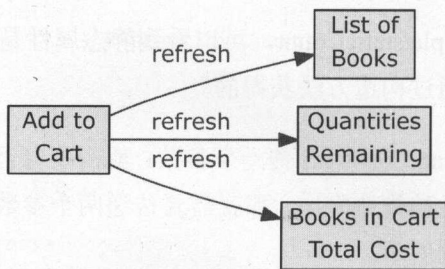


图 2-2 组件之间的直接引用

当使用观察者时, “加入购物车”组件只需要通知给观察者, 页面上其他绑定在观察者上的组件就会监听这种变化并且自动更新以显示最新的数据。正如图 2-3 所示, “加入购物车”组件不再绑定在其他组件上。只需要通知观察者更新, 其他监听观察者的组件就会察觉这种变化。如此一来, “加入购物车”组件就可以保持独立并且不需要知道页面上的其他组件。

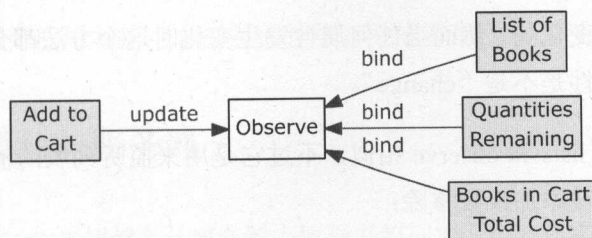


图 2-3 观察者保证组件之间是解耦的

以下例子是如何创建一个观察者并且让它监听属性的变化：

```
canjs/public/concepts/concepts-test.js
```

```
// Create an observe
```

```
var observe = new can.Observe({});
```

```
// Listen for changes on the "title" attribute
```

```
observe.bind("title", function(evt, newTitle, oldTitle) {
  console.log("title: newTitle=", newTitle, "oldTitle=", oldTitle);
});
```

```
// Set a value for the "title" attribute
```

```
observe.attr("title", "First");
```

```
// the console logs:
```

```
// title: newTitle= First oldTitle= undefined
```

```
// Set another value for the "title" attribute
```

```
observe.attr("title", "Second");
```

```
// the console logs:
```

```
// title: newTitle= Second oldTitle= First
```

正如你看到的那样，属性是由 `attr` 方法来设置，并使用属性名和属性值作为参数的。可以只提供属性名称来获取值，比如说 `observe.attr("title")`。

另外，还可以通过绑定来监听任何属性的变化：

```
canjs/public/concepts/concepts-test.js
```

```
observe.bind("change", function(evt, attr, how, newValue, oldValue) {
  console.log("change: attr=", attr, "how=", how,
    "newValue=", newValue, "oldValue=", oldValue);
});
```

```
observe.attr("title", "Third");
```

```
// change: attr= title how= set newValue= Third oldValue= Second
```

```
observe.removeAttr("title");
```

```
// change: attr= title how= remove newValue= undefined oldValue= Third
```

如果你真的有一个属性名字叫作 “change”，则还是可以通过调用 `bind("change", ...)`

来监听这个属性的变化的。然而当任何属性发生变化时这个方法都会被调用，所以你必须检查变化的属性是不是“change”。

最后，observe lists 和 observe 相似，不过它是用来监听列表的值的。使用它可以监听列表中值是否被添加或被移除：

```
canjs/public/concepts/concepts-test.js
```

```
var observe = new can Observe.List([42, 44, 46]);
observe.bind("add", function(evt, newValues, index) {
  console.log("add: newValues=", newValues, "index=", index);
});
observe.bind("remove", function(evt, oldValues, index) {
  console.log("remove: oldValues=", oldValues, "index=", index);
});

observe.push(48);
// add: newValues= [48] index= 3
observe.splice(1, 2);
// remove: oldValues= [44, 46] index= 1
```

在下面的章节中，你会看到更多关于如何让观察者为我们工作的例子。

2.2.4 使用 CanJS 创建一个前端书签应用

下面，我们将会为第一章 Sinatra 第一页中创建的书签服务器应用创建一个单页面 JavaScript 前端。我们已经使用 Mustache 模板创建了一个简陋的用户界面，现在要做一些更酷炫的东西，可以动态刷新页面而不用全页面加载。CanJS 是一个只有客户端的框架，可以和 REST 以及 JSON 接口很好地合作（虽然并不一定需要 REST/JSON 服务器）。同样地，它和 Sinatra 服务器完美契合。图 2-4 所示的截屏展示了应用完成以后的样子。

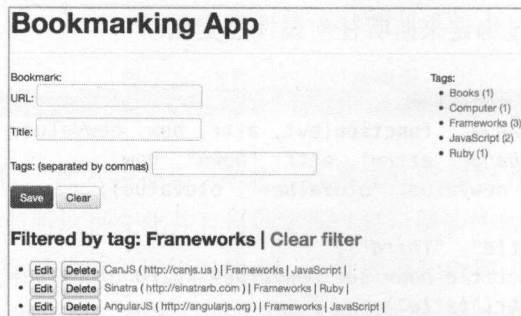


图 2-4 完成后的 CanJS 书签应用

首先我们为这个应用创建一个书签的模型，让它和服务器交互。

2.2.5 连接模型与服务器

我们要做的第一件事情就是从服务器上取得书签列表，同时还需要考虑保持和服务
器上数据的同步。我们可以绑定监听器，监听器在属性变化时就发送 Ajax(Asynchronous
JavaScript and XML、异步 JavaScript 和 XML)请求，并且会处理相应的服务器响应来
更新属性。

我们还可以使用 `can.Model`。`can.Model` 是基于 `can.Observe` 创建的，并且可以让
你来指定数据与服务器交互的方式。下面是一个书签模型和 Sinatra 实现的书签服务
器交互的例子：

```
canjs/public/app/base/app.js
var Bookmark = can.Model.extend({
  findAll: "GET /bookmarks",
  create: "POST /bookmarks",
  update: "PUT /bookmarks/{id}",
  destroy: "DELETE /bookmarks/{id}",
}, {
});
```

我们已经为服务器定义了创建、读取、更新和删除书签的应用程序接口 (Application
Program Interface, API)。`CanJS` 可以自动将 `findAll`、`create`、`update` 和 `destroy` 属性转
换成方法，发送 Ajax 请求并且使用那些与属性相关联的字符串来处理响应。字符串
的第一部分是请求的方法 (GET, POST, PUT, DELETE)，第二部分是请求的统一资源
标识符 (Uniform Resource Identifier, URI)。URI 里任何在 `{}` 中的部分都会被对象中
相应的属性值替换。比如，要想更新当前已有的一个 `Bookmark` 对象，`CanJS` 首先会
调用 `bookmark.attr("id")`，将取得的值替换掉 URI 中 `{id}` 的部分，最后会发送一个 PUT
请求。

模型还会处理服务器的响应。比如 `Get` 请求，`CanJS` 会将响应解析成 JSON，
并且会使用相同的属性值创建模型对象。而对于 `PUT` 和 `POST` 请求，模型对象的
属性会被装进请求体里发送到服务器，而服务器返回的那些属性值又会被对应地
更新到模型中去。特别是对于 `POST` 请求的响应，服务器应该返回新创建的对应
的 ID 值和其他一些新建的或者更新的属性值，而没有被改变的属性值就不需要被

返回了。对于 DELETE 请求，对象只需要检查服务器有没有返回一个成功的标示代码。

在 `can.Model` 的对象中，还有一个 REST 方法是 CanJS 会自动处理的。这里没有提到是因为我们的书签管理系统没有用到它，你们猜是什么？

如果你们的答案是 `findOne`，那么快去冰箱里拿一个好吃的奖励一下自己。没错，这个方法就是用来通过 ID 值从服务器那里获取一个单独的模型对象的。对于我们的书签对象，就应该是 `findOne`：“GET/bookmarks/{id}”。

为了使用这些方法，我们可以调用 `Bookmark` 对象的 `findAll` 和 `findOne` 方法，并且由回调函数返回查找出的所有书签实例或者一个单独的书签实例：

```
canjs/public/app/base/app.js
Bookmark.findAll({}, function(bookmarks) {
});

Bookmark.findOne({id:42}, function(bookmark) {
});
```

请注意 `findAll` 和 `findOne` 的第一个参数是一个对象，可以和其他参数一起随请求发送出去。

创建或更新一个对象，可以调用模型实例里的 `save()` 方法，会自动发送一个 POST 或 PUT 请求。如果一个模型实例没有 `id` 属性，那它就是一个新的对象；如果有 `id` 属性，那它就是一个已存在的对象。这样 CanJS 就知道应该相应地调用 `create` 方法还是 `update` 方法了。最后，对一个模型实例调用 `destroy` 方法会发送一个 DELETE 请求，并且把对象的 `id` 传过去。

在处理模型列表方面，`can.Model.List` 和 `can Observe.List` 一样也有一个额外的特性：在调用了 `destroy()` 方法后，`can.Model.List` 可以自动将这个对象从列表中移除。

我们已经可以从服务器端取得书签数据了，那么该如何展示它们呢？

2.2.6 渲染视图

CanJS 使用 Mustache 或者 EJS 来渲染视图，在这里我们会使用 Mustache。那么，

还记得我们在 Mustache 那一章看到的 Mustache 语法吗？

这也是一个 Mustache 的模板，我们用它来显示一个书签的列表，每一个书签项都有一个编辑和删除按钮。

```
canjs/public/app/base/bookmark_list.mustache
```

```
<ul>
  {{#bookmarks}}
    <li>
      <button class="edit">Edit</button>
      <button class="delete">Delete</button>
      <a href="{{url}}">{{title}}</a>
      ( <a href="{{url}}">{{url}}</a> )
    </li>
  {{/bookmarks}}
</ul>
```

想要渲染这个模板，可以通过传递其文件的 URI 来调用 `can.view` 方法，而无需使用 `.mustache` 的后缀。注意，这个文件的路径是 `public/app/base/bookmark_list.mustache`。

记住，Sinatra 只识别放在 `public` 路径下的文件。调用模板的 URI 是 `/app/base/bookmark_list`。

另一个传给 `can.view` 的参数，是需要通过模板展现的模型数据。把它们都放在一起，我们就得到了这个：

```
canjs/public/app/base/app-test.js
```

```
// a list of bookmarks, as we would receive from the server
var bookmarks = [
  {url:"http://one.com", title:"One"},

  {url:"http://two.com", title:"Two"}
];

var viewModel = {bookmarks:bookmarks};
var element = $("#target");

// Render view by calling can.view
element.html(can.view("/app/base/bookmark_list", viewModel));

// can.view is implicitly called
element.html("/app/base/bookmark_list", viewModel);
```


注意，我们调用了 `can.view` 并把结果通过 jQuery 的 `html` 方法添加到页面元素上去；也可以使用模板的路径和视图对象直接调用 `html` 方法，CanJS 就会忠实地为你调用 `can.view`。这对其他的 jQuery 方法也适用，比如 `append`、`prepend`、`after` 等。

2.2.7 动态绑定

当传给视图的对象是观察者，CanJS 会自动地做动态绑定。这样当观察者的属性发生改变时，就会自动更新视图。这个特性对于观察者列表同样适用，当有对象被添加进列表或被从列表中移除时，列表的视图也会相应地更新。让我们看看下面这个例子：

```
canjs/public/app/base/app-test.js
// 'bookmarks' is now a list of observes
var bookmarks = new can.Observe.List([
  {url:"http://one.com", title:"One"},
  {url:"http://two.com", title:"Two"}
]);
var viewModel = {bookmarks:bookmarks};
$("#target").html("/app/base/bookmark_list", viewModel);

// The view automatically refreshes to display these changes
bookmarks[0].attr("title", "Uno");
bookmarks.push({url:"http://three.com", title:"Three"});
```

依靠动态绑定的特性，你就可以只关注与模型相关的工作，而把更新视图的任务交给 CanJS 去做。由于 `can.Model` 继承自 `can.Observe`，所以你通过 Ajax 调用（比如 `findAll`），获得的对象都是观察者。你可以直接以它们作为你的模板的视图模型，同时享受动态绑定带来的好处。

2.2.8 我们在第1天学到的

今天我们一起学习了 CanJS 的几个关键部分：`can.Construct`、`can.Observe`、`can.Model` 和 `can.view`。

我们学会了如何创建对象的层次关系和继承的使用。我们学习了观察者特性是如何让应用的各个部分相互独立的，还学习了如何创建一个与服务器同步的模型，并且

学习了如何将它显示在视图上。CanJS 还有一个重要的部分就是 `can.Controller`，我们明天将会学习到。

我们已经讨论过 CanJS 的核心原则：保持组件之间的相互独立和在组件与模型之间使用观察者模式来通知变化从而实现与服务器端的数据同步。最后我们学习了如何通过使用视图中的观察者对象来实现动态绑定的特性，从而不需要进行手动刷新。

第1天的自学

查阅

- 主要的 CanJS 论坛。
- CanJS 的 API 文档。
- 用 CanJS 实现的 TodoMVC。

实践

- 实现一个简单的带有观察者模型的 CanJS 页面，在浏览器的控制台中使用 JavaScript 改变观察者模型的属性，看看视图是如何自动更新的。
- 在 jsFiddle 上使用 Canjs jQuery Template 试验 CanJS 的使用¹。

2.3 第2天：创建控制器

我们现在已经创建好了模型和视图，当模型发生变化时视图也会自动更新。那么，如何将这二者结合成为一个组件呢？当我们点击书签列表上的“编辑”或者“删除”按钮时，希望可以编辑或者删除对应的书签。这就需要事件处理的代码来起作用，即 CanJS 中所说的控制器。今天我们来一起学习一下 `can.Control` 的四个主要部分。

1. 将控制器绑定到页面元素上去。

¹ <http://jsfiddle.net/donejs/qYdwR/>

2. 监听用户界面事件。
3. 使用 `data()` 方法从页面取回模型数据。
4. 使用观察者实现控制器之间的通信。

为了做到这一点，我们需要创建两个控制器：一个负责管理书签列表，另一个负责管理创建和编辑书签列表的表单。最后，我们将得到书签应用的第一版。

2.3.1 将控制器绑定到页面元素上

要创建一个控制器，我们需要传递两个参数给初始化方法：一个是你想要绑定到控制器上去的元素；另一个是一个选项对象，它可以包含你需要的任何附加的参数。下面是一个例子：

```
canjs/public/concepts/concepts-test.js
var MyControl = can.Control.extend({
  init: function(element, options) {
    var view = "/concepts/bookmarks";

    element.html(view, {bookmarks:options.bookmarks});
  }
});

var bookmarks = []; // this would normally be the real list of bookmarks
var options = {bookmarks:bookmarks};
new MyControl("#bookmark_container", options);
```

注意，这个元素是一个 jQuery 选择器。在前一个例子中，控制器与页面上 id 为 “bookmark_container” 的元素绑定在一起。虽然传过来的参数为字符串，CanJS 还是可以自动将其转化为相应的 jQuery 元素，从而可以在控制器的初始化方法里调用元素上的方法。我们在这里就是调用了 `element.html(...)`。

当然你也可以传递真正的 jQuery 对象，而不是一个字符串。

```
var element = $("#bookmark_container");
new MyControl(element, ...);
```

如图 2-5 所示，控制器绑定在页面元素上。

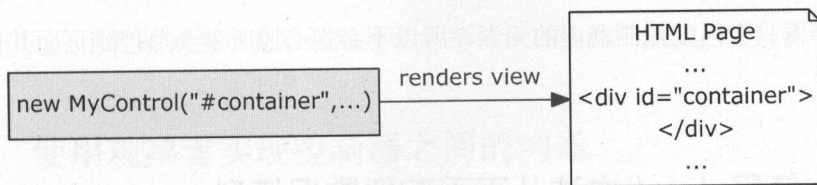


图 2-5 控制器绑定在元素上

控制器将视图转化为页面元素，并且将自己限制在元素范围内。也就是说，控制器只监听它所绑定的元素上面发生的 UI（User Interface 用户界面）事件。

在初始化方法中，你可以使用通过选项对象传过来的数据，将控制器上的元素渲染成视图。控制器的另一个功能可以让你通过 `this.element` 和 `this.options` 来获取元素和选项对象。这些都是由 CanJS 自动设置好的。

2.3.2 监听 UI 事件

我们已经创建好了一个控制器，并且使用它将视图渲染成页面元素。下一步就该处理 UI 事件了，比如用户点击按钮、选中链接等。为了做到这一点，我们需要在控制器中定义一个字符串型的属性以及它相应的方法。

```

"selector eventType": function(el, evt) {
  // el is the element on which the event occurred
  // evt is the event object
}
  
```

我们通过 jQuery 选择器获取对应的元素，并给这个元素绑定指定事件类型的监听器。

```

canjs/public/concepts/concepts-test.js
var MyControl = can.Control.extend({
  // Listen for click events on buttons
  "button click": function(el, evt) {
    // ...
  },
  // Listen for change events on checkboxes under elements with class="item"
  ".item :checkbox change": function(el, evt) {
    // ...
  }
});
  
```


选择器只会匹配控制器内的元素，所以不必担心监听器会监控到页面其他部分元素的事件。

2.3.3 使用 data()方法从页面获取数据模型

当监听 UI 事件的时候，handler 方法可以接收到元素和事件对象。比如，用户会点击书签旁边的“编辑”或者“删除”按钮。而我们真正需要的是书签模型对象，如何获取它呢？

首先使用 data Mustache heler 将数据模型对象与视图上的元素绑定，`{{data "name"}}`。其中，“name”可以是任何一个你想要获得的关于数据模型的名称。在书签列表视图中，书签模型对象和``元素绑定：

```
canjs/public/app/base/bookmark_list.mustache
```

```
<ul>
  {{#bookmarks}}
  > <li {{data "bookmark"}}>
    <button class="edit">Edit</button>
    <button class="delete">Delete</button>
    <a href="{{url}}">{{title}}</a>
    ( <a href="{{url}}">{{url}}</a> )
  </li>
  {{/bookmarks}}
</ul>
```

想要获取到模型对象，就需要在控制器里调用 `element.data("name")`，并且模型对象必须含有 `{{data, "name"}}`。在书签列表里，当用户点击按钮时，在事件响应方法内的元素才是真正的`<button>`元素。我们可以通过 jQuery 的 `closest` 方法，`element.closest("li")`，来获取它的``父元素。最后，我们通过 `data("bookmark")` 来获得书签对象：

```
canjs/public/app/base/app.js
```

```
// retrieve the bookmark object from the <li> parent element
getBookmark: function(el) {

  return el.closest("li").data("bookmark");
},

// handle the click on the delete button, destroy the bookmark
".delete click": function(el, evt) {
  this.getBookmark(el).destroy();
},
```

使用 `data` 方法从视图层获取对象是非常简单和直接的手段。

2.3.4 使用观察者实现控制器之间的沟通

使用观察者可以保证控制器之间进行事件通信的同时解耦合，这通常也被称为“状态总线”或者“事件总线”。如图 2-3 所示，观察者可以保证组件之间的解耦。有观察者作为组件之间通信的桥梁，组件彼此之间就不需要直接引用了。

想要触发观察者的一个事件，可以调用 `call.trigger(eventHub, "eventType", data)`。把观察者和我们之前学的所有东西都融入书签列表的控制器里，就是以下代码：

```
canjs/public/app/base/app.js
```

```
var BookmarkListControl = can.Control.extend({
  view: "/app/base/bookmark_list",

  init: function(element, options) {
    // save a reference to the eventHub observe
    this.eventHub = options.eventHub;
    // render the view on the element with the bookmarks as the model
    var view = options.view || this.view;
    element.html(view, this.getViewModel(options));
  },

  getViewModel: function(options) {
    return {bookmarks:options.bookmarks};
  },

  // retrieve the bookmark object from the <li> parent element
  getBookmark: function(el) {
    return el.closest("li").data("bookmark");
  },

  // handle the click on the delete button, destroy the bookmark
  ".delete click": function(el, evt) {
    this.getBookmark(el).destroy();
  },

  // handle the click on the edit button, trigger an editBookmark event
  ".edit click": function(el, evt) {
    can.trigger(this.eventHub, "editBookmark", this.getBookmark(el));
  }
});
```

如你所见，我们通过在事件总线对象上触发一个 `editBookmark` 事件来处理“编辑”按钮的行为。书签列表不需要知道哪个对象与编辑书签有关，而且监听 `editBookmark` 事件的组件也不需要知道是应用的哪个部分触发了这个事件。

想要使用我们创建的 `BookmarkListControl`，就要把想要关联的页面元素、包含了事件总线观察者的选项对象以及书签列表传递给它。

```
canjs/public/app/base/app.js
var App_base = can.Construct.extend({
  init: function() {
    // Retrieve the bookmarks from the server
    Bookmark.findAll({}, function(bookmarks) {
      // Create the event hub observe
      var eventHub = new can.Observe({});
      // Create the options object with the event hub and the bookmarks
      var options = {eventHub:eventHub, bookmarks:bookmarks};

      // Create the control, attaching it to the element on the page
      // that has id="bookmark_list_container"
      new BookmarkListControl("#bookmark_list_container", options);

      // Create the bookmark form control (which we build in the
      // next section.)
      new BookmarkFormControl("#bookmark_form_container", options);
    });
  }
});
```

我们已经使用书签控制器构建了整个应用，注意代码里还包含了书签表单控制器。下一章我们会谈到如何创建一个表单控制器，但现在先使用样例代码运行一下试试。可以简单地调用 `new App_base()` 来启动应用。让我们在一个包含了控制器对应的页面元素的 HTML 主页面里来做这件事：

```
canjs/views/index.mustache
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Bookmarking App</title>
  </head>

  <body>
    <h1>Bookmarking App</h1>
    <hr>
    > <div id="bookmark_form_container"></div>
```

```

➤ <div id="bookmark_list_container"></div>
  </body>

  <script src="/lib/jquery/jquery-1.10.2.js"></script>
  <script src="/lib/canjs.com-1.1.8/can.jquery.js"></script>
  <script src="/lib/canjs.com-1.1.8/can.construct.super.js"></script>
  <script src="/lib/canjs.com-1.1.8/can.observe.validations.js"></script>
  <script src="/lib/canjs.com-1.1.8/can.observe.list.js"></script>
  <script src="/lib/canjs.com-1.1.8/can.view.modifiers.js"></script>
  <script src="/lib/canjs.com-1.1.8/can.mustache.js"></script>
  <script src="/app/base/app.js"></script>
➤ <script>new App_base();</script>
</html>

```

也可以通过启动配置在 canjs 路径下的 Sinatra 服务器来启动应用。

```
$ ruby app.rb
```

在浏览器中打开 <http://localhost:4567>，应该可以看到一个书签应用的初始版本，如图 2-6 所示。你可以尝试创建、更新和删除书签，来看看书签列表是如何自动刷新的。

图 2-6 书签应用初版

让我们继续学习如何创建一个书签表单控制器。

2.3.5 创建一个表单控制器

我们需要创建一个表单来创建和更新书签，这个表单需要文本输入框来让用户输

入 URL 和标题。创建表单的视图模板非常直观：

```
canjs/public/app/base/bookmark_form.mustache
Bookmark:
<form action="/bookmarks" method="post">
  <label>
    URL:
    ❶ <input type="text" name="url" value="{{url}}">
  </label>
  <label>
    Title:
    <input type="text" name="title" value="{{title}}">
  </label>
  ❷ <button class="save btn btn-primary" {{data "bookmark"}}>Save</button>
  <button class="clear btn">Clear</button>
</form>
```

❶ 表单使用书签作为数据模型。当编辑书签对象时，表单上的 url 和标题输入框将书签模型对应属性的值取出并显示出来。

❷ 书签和“保存”按钮通过一个 data helper 关联起来。当我们处理“保存”按钮的点击事件时，可以轻易地获取书签对象的内容。

下面让我们一起创建书签表单的控制器。首先创建一个空的控制器，并填充相应的属性：

```
canjs/public/app/base/app.js
var BookmarkFormControl = can.Control.extend({
  // Add properties here.
});
```

第一步我们需要创建一个属性“Bookmark”来对应书签模型，这样就可以在 Bookmark FormControl 的子类中改变书签的内容了。同样的，我们还需要一个属性“view”来对应视图模板。

```
canjs/public/app/base/app.js
BookmarkModel: Bookmark,
view: "/app/base/bookmark_form",
```

下一步，让我们来添加控制器的初始化方法和书签的编辑方法：

```
canjs/public/app/base/app.js
init: function(element, options) {
  ❶ this.BookmarkModel.bind("created", function(evt, bookmark) {
```

```

    options.bookmarks.push(bookmark);
  });
  this.clearForm();
},
2 editBookmark: function(bookmark) {
    var view = this.options.view || this.view;
    this.element.html(view, bookmark);

    bookmark.bind("destroyed", this.clearForm.bind(this));
  },
3 clearForm: function() {
    this.editBookmark(new this.BookmarkModel());
  },
4 "{eventHub} editBookmark": function(eventHub, evt, bookmark) {
    this.editBookmark(bookmark);
  },

```

① 通过和书签模型的“created”事件绑定，我们可以看到书签是在何时由表单创建的，以及我们将书签加入书签列表的时间。

② 我们通过加载以书签对象作为模型的视图，来将书签对象与表单关联起来。然后当我们清空表单时，也可以监听到书签对象的销毁。这就实现了用户编辑一个表单然后决定将它删除的场景。

③ 为了清空表单，我们将一个新的空书签对象绑定在表单上。

④ 还记得我们是在怎样在书签列表控制器上触发一个“编辑书签”事件的吗？我们就是这样在表单控制器中监听这个事件的。{eventHub} 语法监听了在 options.eventHub 对象上的所有事件，这个方法可以获得事件发生时的相关对象、事件对象和随着事件传递的所有数据。在该例子中，这个数据就是要被编辑的书签。我们只是简单地使用 editBookmark 方法来将书签与表单绑定。

同时，我们还会添加处理器来保存书签：

```
canjs/public/app/base/app.js
```

```

".save click": function(el, evt) {
    evt.preventDefault();
    var bookmark = el.data("bookmark");
1 bookmark.attr(can.deparam(el.closest("form").serialize()));
    this.saveBookmark(bookmark);
  },
  saveBookmark: function(bookmark) {
2 bookmark.save(this.clearForm.bind(this), this.signalError);
  },

```

```

},
signalError: function() {
  alert("The input is not valid.");
},

```

① 当保存书签时，我们可以获得书签对象以及它的数据方法。我们需要将相应的数值从用户填写的表单中获取出来，然后填入对象的对应属性中去。通过调用表单对象的序列化方法以及调用 `can.deparam` 就可以做到这一点，而不需要手动地轮循表单对象的每一个输入框。这样做会生成一个对象，我们可以通过这个对象将表单输入框中的值对应地传到表单对象的属性中去。可以这么做是因为表单的每一个输入框的名字都和表单对象的属性名称一一对应。

② 我们通过调用 `save` 方法来保存书签。`save` 方法有两个参数，分别是成功和失败的回调方法。在成功的回调方法里，我们清空表单，以便被下一个书签对象使用；在失败的回调方法里，我们会通知用户。

最后，处理“clear”按钮的事件，清空表单：

```

canjs/public/app/base/app.js
".clear click": function(e1, evt) {
  evt.preventDefault();
  this.clearForm();
}

```

给应用加书签表单控制器的流程和加表单列表控制器是一样的——在页面上创建一个元素，实例化这个控制器，将页面元素和选项对象传递过去。

```

canjs/views/index.mustache
<div id="bookmark_form_container"></div>

```

```

canjs/public/app/base/app.js
new BookmarkFormControl("#bookmark_form_container", options);

```

恭喜，书签应用的第一个迭代结束了，一起来试用一下吧。

2.3.6 我们在第2天学到的

今天我们所学的东西都和 CanJS 控制器相关。我们学习了如何将数据对象的视图

与控制器相结合并绑定到页面元素上去；控制器在监听器的方法里处理 UI 事件，并且可以方便地获取模型数据。我们遵循“创建的模块需要相互独立”的原则来创建部件，用它们来改变模型的属性，以及触发监听器上的事件。

在第一个迭代中，我们还实现了书签应用的前端同时实现了一个可以新建和编辑表书签的表单、一个可以和服务器同步的模型、一个可以自动更新的列表。第2天将会学习与模型交互的其他方式，比如校验、打标签、过滤等。

第2天的自学

查阅

- 查看模板事件的处理文档——CanJS 控制器的另一种事件监听方式。
- 查看一个在 `can.Control.prototype` 里调用了 `on()` 方法来将事件处理器重新绑定到其他模型对象上的例子

实践

- 试用本书中书签应用的源代码。
- 解释为什么我们需要在 `editBookmark` 的 `bookmark.bind("destroyed", this.clearForm.bind(this))`；中调用 `bind(this)` 函数。如果我们写成 `bookmark.bind("destroyed", this.clearForm)`；会怎么样？
- 改变书签视图的渲染样式，比如用图标来替换“编辑”和“删除”的文字。这样会改变书签列表控制器中的事件渲染机制吗？

2.4 第3天：与模型的协作

在第三天，也是最后一天 CanJS 的学习中，我们将一起探索更多与观察者以及模型对象协作的方式。要记住这两个是 CanJS 应用的重点：驱动模型并使视图自动更新，在观察者上触发事件并在控制器上进行事件处理。让我们一起来看看如何根据不同的需求塑造模型：通过创建过滤列表和添加帮助方法。首先我们来看看如何在书签模型上做校验。

2.4.1 添加校验

在 Sinatra 的 Web 应用中，每一个书签都必须有一个标题和 URL，并且 URL 的格式也需要被检查。服务器端的检验固然很好而且很有必要，但是客户端的校验可以给用户更直接迅速的反馈，而且可以节省不必要的通信。

我们可以通过在客户端调用校验方法来为书签模型添加校验：

```
canjs/public/app/validation/app.js
// Extend the base Bookmark model
var ValidatingBookmark = Bookmark.extend({
  init: function() {
    var urlPattern = new RegExp(
      "(http|https):\\|\\|/(\\|w+:{0,1}\\|w*@)?(\\|S+)(:[0-9]+)?" +
      "(\\|\\|\\|/(\\|\\|w#?!:..?+=&%@!\\|\\|\\|/))?)");
    // Add validations
    this.validatePresenceOf(["url", "title"]);
    this.validateFormatOf("url", urlPattern);
  }
}, {
});
```

调用 `validatePresenceOf` 时需要 URL 和标题。URL 的格式通过 `validateFormatOf` 和一个正则表达式来校验，其他的校验方法如下。

- `validateInclusionOf`：将属性限制在一组合法的值里。
- `validateLengthOf`：设置了一个属性的最大和/或最小长度。
- `validateRangeOf`：设置了一个属性的最大和/或最小数值。
- `validate`：设置了一个自定义校验条件——你可以创建一个可以接收校验值和待校验的属性名称的校验方法，当校验成功时这个方法返回 `null`，而失败时则返回错误信息。

有了这个，我们就可以在模型实例上进行校验了。调用 `errors()` 会返回一个键值对形成的表，键名为属性名称，值为错误信息。还可以使用 `errors(attrName)` 来指出出错的属性名称，而不是给出一堆错误信息。在这两种情况下，当对象被校验成功，校验方法都不会返回任何值。

想要实现一个带校验的书签表单控制器，我们需要先扩展它的父类控制器：

```
canjs/public/app/validation/app.js
// Extend the base bookmark form control
var ValidatingBookmarkFormControl = BookmarkFormControl.extend({
  // Add properties here.
});
```

我们要做的第一件事情，就是覆盖书签模型来使用校验方法。

```
canjs/public/app/validation/app.js
BookmarkModel: ValidatingBookmark,
```

下一步，我们覆盖 `editBookmark` 方法：

```
canjs/public/app/validation/app.js
editBookmark: function(bookmark) {
  this._super(bookmark);
  var self = this;
  bookmark.bind("change", function() {
    var errorMessage = bookmark.errors() ?
      can.map(bookmark.errors(), function(message, attrName) {
        return attrName + " " + message + ". ";
      }).join("")
      : "";
    self.element.find(".text-error").html(errorMessage);
  });
},
```

在父类调用 `editBookmark` 方法之后，我们使用 `bind` 方法来监听属性的变化。当属性发生变化时我们进行校验，如果有错就调用 `errors()` 方法并且创建一条错误信息，如果没错就将错误信息留空。然后我们可以将错误信息显示在页面的 `.text-error` 元素上。最后我们覆盖 `saveBookmark` 方法，使它在保存书签时先进行校验。

```
canjs/public/app/validation/app.js
saveBookmark: function(bookmark) {
  if (!bookmark.errors()) {
    this._super(bookmark);
  }
}
```

在书签表单模板中，我们将 `.text-error` 元素添加在表单的最后一个输入框之后：

```
canjs/public/app/base/bookmark_form.mustache
<label>
  Title:
  <input type="text" name="title" value="{{title}}">
</label>
➤ <div class="text-error"></div>
```

当 URL 或者标题不合法时, 就会出现一个如图 2-7 所示的错误提示信息。

图 2-7 显示验证错误信息

现在我们拥有一个完美的带校验的表单了, 接下来进一步完善应用吧。我们的 Sinatra 服务器可以处理书签与标签之间的联系, 接下来就给书签表单添加一个输入标签的输入框。

2.4.2 实现标签的处理

服务器为每一个书签都返回一个名为 `tagList` 的属性, 它就是一个标签的列表。想要输入一组标签, 用户只需要输入一串由逗号隔开的字符串即可。我们可以在书签模型上创建一个 `tagsAsString` 属性, 然后如下所示来回操作 `tagList`:

```
canjs/public/app/tagfilter/app.js
// Extend the validation bookmark model
var TaggedBookmark = ValidatingBookmark.extend({
  init: function() {
    // Initialize tagsAsString from tagList
    var tagList = this.attr("tagList");
    this.attr("tagsAsString", tagList.join(", "));

    // Listen for changes on tagsAsString and set tagList
    this.bind("tagsAsString", this.onTagsAsStringChange);
  },
  onTagsAsStringChange: function(evt, tagsAsString) {
    // Split the string by comma and trim whitespace
    var trimmed = can.map(tagsAsString.split(","), can.trim);

    // Ignore empty tags, for example if the user entered a,,,b
    var byNotEmpty = function(tag) {
      return tag.length > 0;
    };
    var notEmpty = can.filter(trimmed, byNotEmpty);
    var tagList = this.attr("tagList");
```

```
// Update the tag list to match the ones entered by the user
tagList.attr(notEmpty.sort(), true);
}
});
```

我们已经为书签模型添加了新的属性：tagsAsString。在实例化的时候，由 tagList 为它赋初始值。通过与事件处理器的绑定，当 tagsAsString 变化时 tagList 的值也会被重新设定。

现在可以给书签表单模板添加一个文本输入框，然后将这个输入框和 tagsAsString 绑定：

```
canjs/public/app/tagfilter/bookmark_form.mustache
```

```
<label>
  Title:
  <input type="text" name="title" value="{{title}}">
</label>
> <label>
>   Tags: (separated by commas)
>   <input type="text" name="tagsAsString" value="{{tagsAsString}}">
> </label>
```

由于书签表单控制器与所有的表单输入框绑定，并且与模型属性一一对应，我们保存标签时就不需要再做任何别的事情了。记住我们使用了 attr，can.deparam 和 serialize 来实现表单和书签模型之间值的绑定：

```
bookmark.attr(can.deparam(el.closest("form").serialize()));
```

在书签列表中显示标签也非常容易：

```
canjs/public/app/tagfilter/bookmark_list.mustache
```

```
<ul>
  {{#bookmarks}}
    <li {{data "bookmark"}}>
      <button class="edit">Edit</button>
      <button class="delete">Delete</button>
      <a href="{{url}}">{{title}}</a>
      ( <a href="{{url}}">{{url}}</a> ) |
    </li>
    {{#tagList}}
      <a class="tag" href="#" {{data "tag"}}>{{this}}</a> |
    </li>
  {{/bookmarks}}
</ul>
```


每一个书签的旁边，都有一个简单的循环来显示所有的标签。所有的这些标签都是可点击的，这些链接都含有一个`{{data "tag"}}`的引用与标签相关联。下一个很棒的特性是，当用户点击标签时可以过滤书签。让我们来试试吧！

2.4.3 过滤书签

CanJS 的 `can Observe.List` 里的过滤方法可以很容易地实现过滤的功能，可以通过加载 `can/observe/list/js` 来获得。通过调用过滤器然后传一个可以返回 `true` 或者 `false` 的方法来判断应该保留哪个书签，我们可以获得一个过滤后的书签列表。这个书签列表可以传给书签列表控制器。

第一步，创建一个过滤器。它是一个拥有过滤标签和过滤方法的监视器：

`canjs/public/app/tagfilter/app.js`

```
var filterObject = new can.Observe({
  filterTag: "" // the filter tag is initially blank
});
var filterFunction = function(bookmark) {
  var tagList = bookmark.attr("tagList");
  var filterTag = filterObject.attr("filterTag");
  var noFilter = (!filterTag) || (filterTag.length == 0);
  var tagListContainsFilterTag = tagList && tagList.indexOf(filterTag) > -1;
  return noFilter || tagListContainsFilterTag;
};
```

过滤方法可以返回 `true` 或者 `false`，来表明书签是否应该被过滤出来放到结果列表里。当没有过滤标签或者书签的标签列表里含有过滤标签时，该方法就会返回 `true`。

第二步，我们可以通过调用过滤器创建一个过滤后的书签列表，并将它传给过滤方法，然后将这个书签列表传给书签列表控制器：

`canjs/public/app/tagfilter/app.js`

```
TaggedBookmark.findAll({}, function(bookmarks) {
  var eventHub = new can.Observe({});
  // Pass filterObject to the controls
  var options = {eventHub:eventHub, bookmarks:bookmarks,
    filterObject:filterObject};

  // Create the filtered bookmark list
  var filtered = bookmarks.filter(filterFunction);

  // Create an options object with the filtered bookmark list
```

```

var filteredOptions = can.extend({}, options, {bookmarks:filtered});
// ...
// Create the bookmark list control with the filtered bookmark list
new TagFilterBookmarkListControl("#bookmark_list_container", filteredOptions);
// ...
});

```

最后当用户点击标签时，就可以获得过滤后的书签列表。我们只需要继承书签列表控制器，覆盖视图来显示带着链接的书签列表，添加观察者来响应链接的点击事件。监听器获取标签，然后将过滤器设置进过滤器对象中。

canjs/public/app/tagfilter/app.js

```

var TagFilterBookmarkListControl = BookmarkListControl.extend({
  // Use the bookmark list view with the tag links
  view: "/app/tagfilter/bookmark_list",
  // Listen for clicks on tag links, set filterTag on filterObject
  "a.tag click": function(el, evt) {
    var tag = String(el.data("tag"));
    this.options.filterObject.attr("filterTag", tag);
  }
});

```

注意，因为我们使用观察者作为过滤器对象，所以代码还是非常整洁并且解耦的。如图 2-8 所示，控制器只需要将过滤标签设置进过滤器对象。控制器不需要持有书签列表的引用，我们也不需要主动去刷新视图，书签列表会随着过滤器更新。

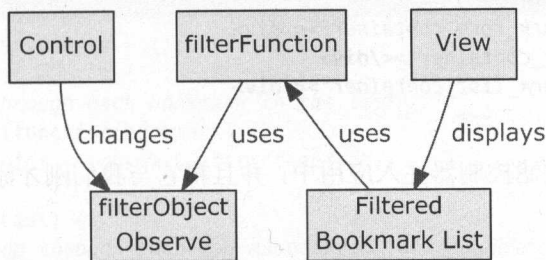


图 2-8 使用观察者过滤书签列表

现在我们可以创建一个标签过滤控制器来展示当前的过滤标签，并且提供一个链接来清除标签以显示所有的书签。这个控制器将过滤模型作为视图模型来支持显示，并且当用户点击清除链接时，控制器会重置过滤标签。

canjs/public/app/tagfilter/app.js

```

var TagFilterControl = can.Control.extend({
  defaults: {

```

```

    view: "/app/tagfilter/tag_filter"
  }, {
    init: function(element, options) {
      this.element.html(options.view, options.filterObject);
    },
    "a.clear click": function(el, evt) {
      this.options.filterObject.attr("filterTag", "");
    }
  });

```

需要再次强调的是，控制器只持有过滤器对象的引用并且只需要重置 `filterTag` 属性来清空过滤器。基于当前的过滤器，视图只显示“过滤结果”和一个清空过滤条件的链接，或者“全部书签”。

```

canjs/public/app/tagfilter/tag_filter.mustache
<h3>
{{#if filterTag}}
  Filtered by tag: {{filterTag}}
  | <a class="clear" href="#">Clear filter</a>
{{else}}
  All bookmarks
{{/if}}
</h3>

```

我们在页面上想要显示过滤控制器的地方添加一个页面元素：

```

canjs/views/index.mustache
<div id="bookmark_form_container"></div>
➤ <div id="filter_container"></div>
<div id="bookmark_list_container"></div>
<!-- ... -->

```

最后，我们将过滤控制器注入应用中，并且将它与我们刚才添加的页面元素联系在一起：

```

canjs/public/app/tagfilter/app.js
TaggedBookmark.findAll({}, function(bookmarks) {
  var eventHub = new can.Observe({});
  // Pass filterObject to the controls
  var options = {eventHub:eventHub, bookmarks:bookmarks,
    filterObject:filterObject};
  // ...
➤ new TagFilterControl("#filter_container", options);
});

```

看，就是这么简单！正如我们所见，控制器给我们在一个自包含的、集中的模块

里组织所有的功能提供了一个很好的方式。现在实现了一个拥有一组标签的书签，也学习了如何通过标签来过滤书签列表。从原始的书签列表数据中，我们还可以创建另一个列表：独特的标签列表，每一个项目显示和这个标签相关联的书签数目。

2.4.4 创建一个标签列表

我们已经学习了如何通过过滤方法创建一个过滤后的书签列表，还可以通过集成 `can.Model.List` 来和列表的数据进行同步。下一步需要做的是遍历书签列表，然后创建一个独特的标签列表，这个列表是由与标签相关联的书签数目组成的。我们会把这个标签列表的数据显示在页面的右边，如图 2-4 所示。到此为止，整个 CanJS 书签应用就完成了。

为了创建一个标签列表，我们循环遍历书签并记录与每一个标签相关的书签的数目。然后将标签排序并返回：

```
canjs/public/app/tagfilter/app.js
```

```
var TaggedBookmark = ValidatingBookmark.extend({
  // ...
});
TaggedBookmark.List = ValidatingBookmark.List.extend({
  // Returns a list of tags, each with label and bookmarkCount
  tags: function() {
    // Keep track of how many bookmarks per tag
    var bookmarkCounts = {};

    // Loop through each bookmark in the list
    this.each(function(bookmark) {
      var tagList = bookmark.attr("tagList");

      if (tagList) {
        // Loop through each tag associated to the bookmark
        tagList.each(function(tag) {
          var existing = bookmarkCounts[tag];
          // Either increase the existing count, or initialize to 1
          bookmarkCounts[tag] = existing ? existing + 1 : 1;
        });
      }
    });

    // The keys in bookmarkCounts are the tag labels
    var labels = Object.keys(bookmarkCounts);

    // Sort the tag labels
```



```

    labels.sort();

    // Return a list of tags with label and bookmark count
    return can.map(labels, function(label) {
        return {label:label, bookmarkCount:bookmarkCounts[label]};
    });
}
});

```

现在，我们已经将标签列表属性加载到书签列表上而不是每一个书签对象上了。这可以让标签列表视图非常简洁：遍历书签列表上的标签对象，将每一个标签的名字和与它相关的书签的数目显示出来：

canjs/public/app/tagfilter/tag_list.mustache

Tags:

```

<ul>
  <!-- Use the dot notation to get the tags attribute -->
  {{#bookmarks.tags}}
    <li>
      <a class="tag" href="#" {{data "tag"}}
        >{{label}} ({{bookmarkCount}})</a>
      </li>
    {{/bookmarks.tags}}
  </ul>

```

标签列表控制器只是负责视图的显示，响应链接的点击事件和设置过滤器上的标签。

canjs/public/app/tagfilter/app.js

```

var TagListControl = can.Control.extend({
  defaults: {
    view: "/app/tagfilter/tag_list"
  }, {
    init: function(element, options) {
      this.eventHub = options.eventHub;
      var model = {bookmarks:options.bookmarks};
      element.html(options.view, model);
    },
    "a.tag click": function(el, evt) {
      var tag = el.data("tag");
      this.options.filterObject.attr("filterTag", tag.label);
    }
  });

```

遍历所有书签拿到所有标签的列表，每一个标签都可以显示有多少书签属于它。由于动态绑定功能，当书签的标签改变时，或有书签被创建或删除时，标签列表和标

签上显示的关联书签的数目都会自动更新。我们来试试吧！

2.4.5 使用路由管理浏览器的位置

当用户点击标签上的链接时，书签列表就会以这个标签作为条件过滤。点击另一个标签，书签列表就会被再次过滤。如果点击浏览器的后退按钮，会怎么样呢？如果可以显示前一次的过滤结果就最好了。让我们来实现这个功能吧！

`can.route`

`can.route` 是与浏览器 hash（也就是 URL 最后以 # 开始的部分）相关联的一个特别的 `can.Observe`。当你设置 `can.route` 的属性时，hash 也会被改变；反之当 hash 改变时，`can.route` 的属性值也会随之改变：

```
canjs/public/concepts/concepts-test.js
location.hash = "#!action=filter"
can.route.attr("action"); // returns "filter"
can.route.attr("tag", "Frameworks");
location.hash; // returns "#!&action=filter&tag=Frameworks"
```

可以监听浏览器的 hash 变化之后，我们就能够创建一个不用刷新整个页面就可以处理浏览器前进后退按钮的单页面 JavaScript 应用了。我们甚至可以将应用的页面标记为浏览器书签。

由于 `can.route` 是一个观察者，所以控制器可以监听它的变化并作出合理的响应。想要监听 `can.route`，需要使用一个特殊的“<pattern>route”语法的字符串作为事件处理的字符串，并且使用一个方法来接收一个单一参数。和我们在 Sinatra 中使用的语法极其相似，这个<patten>是一个 URI 的格式，比如 `filter/:tag`：

```
canjs/public/app/routing/app.js
var RoutingControl = can.Control.extend({
  "filter/:tag route": function(data) {
    this.options.filterObject.attr("filterTag", data.tag);
  }
});
```

这可以处理类似 `#!/filter/Frameworks` 的 hash 值，以及根据给定的标签过滤书签列表。这个事件处理方法可以接受一个单一对象作为参数。这个对象包含了与 URI 中变量对应的参数，也就是那些由冒号开始的部分。

可以看到现在 URI 也非常易读: `#!/filter/Frameworks` instead of `#!/action=filter&tag=Frameworks`。

关于路由有两点非常重要的细节。

- 处理空的 hash (`#`, `#!` 或者没有 hash), 只需要在控制器中使用路由, 不需要 URI 的格式。
- 想要让应用启动的时候强制更新 hash, 需要在控制器的初始化方法里触发一个 `hashchange` 事件。这样可以有效地实现浏览器的加书签功能以及在浏览器的地址栏中直接输入 URL 的功能。

记住上述两点, 然后我们来看看路由控制器:

```
canjs/public/app/routing/app.js
var RoutingControl = can.Control.extend({
  init: function() {
    $(window).trigger("hashchange");
  },
  "route": function() {
    this.options.filterObject.attr("filterTag", "");
  },
  "filter/:tag route": function(data) {
    this.options.filterObject.attr("filterTag", data.tag);
  }
});
```

我们向路由控制器中传递了和向其他控制器传递的一样的选项对象:

```
canjs/public/app/routing/app.js
new RoutingControl(document.body, options);
```

标签列表控制器不再需要 `a.tag` 的点击方法了。点击链接会导致 hash 的变换, 然后被 `RoutingControl` 响应。

一个指向 `#!` 的链接可以清空过滤结果:

```
canjs/public/app/routing/tag_list.mustache
Tags:
<ul>
  {{#bookmarks.tags}}
    <li>
      <a href="#!filter/{{label}}">{{label}} ({{bookmarkCount}})</a>
```

```

    </li>
    {{/bookmarks.tags}}
  </ul>

```

当点击标签时，我们使用了另一种不同的方式来触发过滤事件：只改变 hash 然后让路由控制器来处理这个事件，而不是让控制器监听点击事件再去触发过滤。

```

canjs/public/app/routing/tag_filter.mustache
<h3>
  {{#if filterTag}}
    Filtered by tag: {{filterTag}}
    | <a href="#">Clear filter</a>
  {{else}}
    All bookmarks
  {{/if}}
</h3>

```

路由另一个非常好的特性，就是可以让用户将页面加为浏览器书签。比如用户可能会经常使用一个特定的标签来过滤书签，就可以将这个过滤结果的页面加为浏览器书签页，下一次就可以直接浏览了。

2.4.6 我们在第3天学到的

今天我们学习了 CanJS 观察者和模型的几种用法：我们添加了校验功能、书签列表和过滤功能。最重要的是，我们学习了如何扩展模型，以及添加控制器和视图都可以使用的方法。遵循“更新观察者，刷新一切”的原则，我们构建了一个过滤器对象，任意组件都可以改变这个对象。书签列表的过滤结果可以自动被更新。

最后，我们学习了另一种使用 `can.route` 来触发事件的方法。这让我们可以在单页面应用中管理浏览器导航，添加页面为书签页和手动输入 URL。

第3天的自学

查阅

- 学习 CanJS 中其他可以在你的项目中用得着的公共类和插件。
- EJS——类似于 Mustache，是由 CanJS 提供的另一个模板引擎。

- 学习 `can.compute`，以及它是如何在动态绑定的时候进行值计算的。

实践

- 解释为什么当根据用户输入更新标签列表时，`tagList.attr(notEmpty.sort(), true)` 里需要一个 `true` 标识位。如果忽略这个标识位会怎么样？
- 在书签列表上增加排序的功能，这样就可以将书签列表有序地显示了。
- 通过改变书签列表的视图来改变书签应用的路由版本，从而使用带有 `hash` 的链接。
- 使用 `EJS` 来重做任意一个视图。

2.4.7 对 CanJS 的创造者 Justin B. Meyer 的采访

我们：是什么让你想到要创造 CanJS 的？

Justin: 2006 年 Brian Moschel 和我着手创建一个即需即用服务的平台。使用 GUI，开发人员就可以创建一个 REST（含状态传输的）服务层，并且使用 `JavaScript` 构建 UI。为了增强它，我们创建了 `JavaScriptMVC` 的第一个版本。这可能是第一个支持依赖加载、事件代理、声明动态绑定、RESTful 模型和模板的框架。就在那时，我们公司改变了方向，变成了全职的 `JavaScript` 咨询公司——`Bitovi`。

CanJS 就来自于 `JavaScriptMVC`。由于 `JavaScriptMVC` 的大小成为了 CanJS 的阻碍，我们就单独发布了 CanJS。之后，我们将重心放在创建一个只有主要特性的小而专的内核。

我们：你觉得 CanJS 最棒的特性是什么？CanJS 独特的地方在哪里？

Justin: 这两个问题我会一起回答，因为这两个答案里会有很多重复。

在 `Backbone` 和 `Ember` 的世界里，CanJS 是最好的，而且比这两个都好很多。比如 `Backbone`，对于应用的结构来说它就比较小而且相对简陋。但是 CanJS 就包含了很多 `Ember` 里流行的特性，像动态绑定、值计算。而且 CanJS 的动态绑定更快，且对 `Mustache` 的实现更精准，对值计算的实现也更优雅。

举个例子，在 Ember 里属性的绑定写成这样：

```
var fullName = Ember.computed(function() {  
  return this.get('first') + " " + this.get('last');  
}).property('first', 'last');
```

在 CanJS 里可以写成这样：

```
var fullName = can.compute(function() {  
  return me.attr('first') + " " + me.attr('last');  
});
```

CanJS 能做的不止于此。这个框架着重强调了对内存泄漏的阻止，这一点主要由模板化的事件处理和引用计数模型存储来实现。它可以直接与 jQuery、Zepto、MooTools、YUI 和 Dojo 一起使用，由这些库创建的小组件可以使用 `can.Control` 来绑定。而且，CanJS 还提供了丰富的第一手插件。

我们：你对 CanJS 的未来还有什么规划？

Justin：CanJS 最大的优势之一是 Bitovi 完全致力于这项技术。五年前使用 JavaScriptMVC1.0 的人都可以升级到 CanJS 去使用它所有的特性。CanJS 会持续适配、改进并添加新的功能。

目前 CanJS 有四个举措正在进行，我可以讲讲其中的 3 个。

第一，像 AngularJS 和 Knockout 这样的库让 HTML 为中心的开发再次流行起来。虽然这种方式有一些缺陷——比如它很难在一个元素上组织多个行为——使用作为 Web 组件的控制器理论上讲会更简单一些。对于 1.2 版本，我们打算增加 `can.Component`。它是由一个自定义的元素标签，一个 `can.Control`，一个模板和一个 `can.Observe` 结合在一起的产物。

第二，我们一直致力于开发一个超级模型插件。它会提供一个插入式的客户端缓存，动态绑定和瞬时写入的功能用于很多增删改查的情况¹。

第三，我们已经彻底地改版了 CanJS 的网站和文档。我们打算做主流 JS 框架里拥有最好文档的框架。

¹ <http://bitovi.com/blog/2013/03/weekly-widget-instantaneous-Web-apps.html>

2.5 总结

我们可以看到 CanJS 如何提供了一个非常不错的方式来组织 JavaScript 应用的模型、视图和控制器。模型不仅是一个数据的容器，更通过向应用的其他部分发出信号以保持和服务器的同步来驱动应用。视图保持了 Mustache 语法的简洁，易于阅读，并且可以自动更新来显示模型的变化。控制器将模型和视图结合在一起来处理 UI 事件。

2.5.1 CanJS 的强项

CanJS 在功能性和透明度上达到了完美的平衡。也就是说，你在不牺牲透明度的情况下可以获得最好的特性：你的代码仍然是简单直接的 JavaScript，其中没有任何黑魔法。

这个库非常轻量级，但是又没有牺牲任何特性。它支持了通过向服务器发送 Ajax 请求来进行数据交换，模型和视图间的动态绑定，UI 范围内的控制器事件响应，还有其他一些提供了有用功能的插件。

最后，CanJS 是模块化的：这是一个非此即彼的命题。它的内核可以满足大多数应用的需求，剩下的功能都分布在附加件里，你可以根据自己的需要来添加。

2.5.2 CanJS 的弱项

CanJS 对简单和透明的坚持意味着和使用其他严重依赖与约定和自动化任务的框架相比，需要写更多的代码。这是否受欢迎在于大家的品位，一些开发人员希望用尽可能少的代码完成任务，而另一些就不喜欢框架做太多事情。

2.5.3 最后的思考

这是一个属于 JavaScript 开发人员的时代。我们不乏激动人心的框架，而 CanJS 是其中一个坚实的竞争者。随着新版本的不断推陈出新，开发也是敏捷而充满生机。社区非常友善而且有帮助，很多新人都说学了 CanJS 以后，他们会考虑将它作为自己对框架的选择。

第 3 章

AngularJS

还记得“西蒙说 (Simon Says)”这个游戏吗？发出指令的人说“西蒙说上下跳”，所有的玩家都要上下跳。但是如果扮演西蒙的人发出的指令不是以“西蒙说”开头的，玩家们就不能去做这个动作。这个游戏有趣的地方在于，大家的关注点是在是否执行指令上，而不是在能否完成这个指定动作上。动作是怎样完成的不重要，而是否进行了动作（行动或保持不动）才是能否继续留在游戏中的关键。

AngularJS 就像一个西蒙说 (Simon Says)¹ 游戏一样。作为发出指令的人，你只需描述出做什么，而无须考虑实现的细节。正因如此，AngularJS 应用是以声明方式来实现的。我们上一章讨论的 CanJS 框架是一种更命令式的方法，更关注于执行的细节。哪种方法更好？正如 Vim 和 Emacs 的争辩² 一样，一切由你来决定。能使你的应用最高效的工具就是最好的工具。

3.1 概览

AngularJS 是一个基于“模型-视图-控制器”设计模式 (MVC, Model View Controller) 的前端 JavaScript 框架，与提供 REST/JSON 接口的服务器端搭配起来效果非常好。AngularJS 有着诸多特性，如依赖注入、HTML 指令 (directive) 和自动化

¹ <http://angularjs.org>

² http://en.wikipedia.org/wiki/Editor_war

数据双向绑定。

AngularJS 框架由很多灵活可变的部分组成。在深入了解细节前,我们先从整体看一下。图 3-1 展示了我们即将要探索的 AngularJS 的特性。

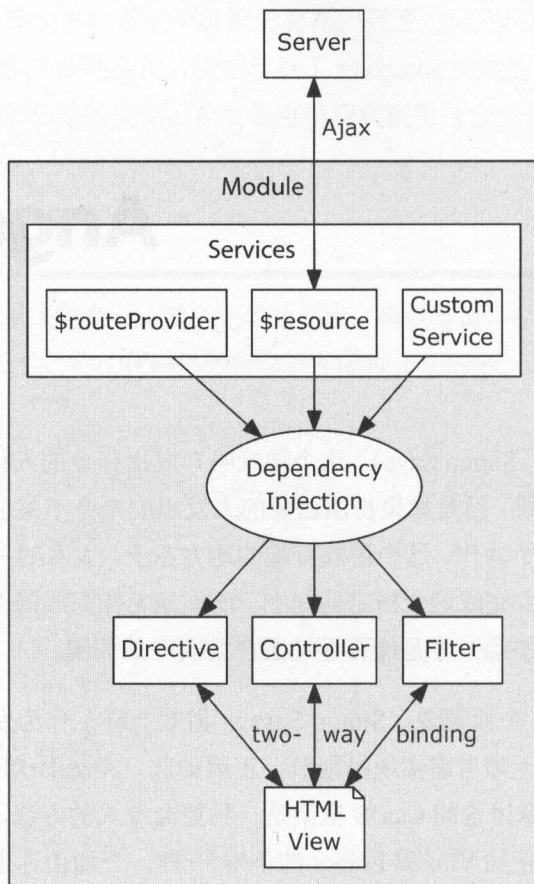


图 3-1 AngularJS 部分概念概述

在 AngularJS 中,模块(module)是整个应用的起始点。模块包含很多组件(component),像服务(service)、控制器(controller)、指令(directive)、过滤器(filter)等。

AngularJS 的依赖注入功能,会自动将代码段或服务整合起来。你可以自定义服务并命名,然后用其名称来指明依赖关系。AngularJS 会将所有服务整合到一起,以轻松地保持代码的松耦合性。

AngularJS 还提供了很多有特色的内置服务,如资源(resource)是一个用来同步

模型（model）和服务端端的内置服务。路由（route）提供者（provider）可以让你通过自定义 URI 达到不刷新整个页面就可以在应用内跳转的效果。

控制器会为视图（view）提供其需要的模型，并为视图提供可以使用的回调函数。它是模型和视图的连接点，同时也是处理视图事件的地方。

指令是 HTML 元素上特殊的属性¹，用来操作文档对象模型（Document Object Model, DOM）并渲染动态数据。AngularJS 有很多内置的指令，当然也可以自定义。AngularJS 还内置了很多过滤器，当然也可以自定义。指令会通过调用它的函数来为视图过滤数据。

自动化数据双向绑定功能是 AngularJS 的另一个亮点，会在模型改变的时候自动重绘相关视图。同样，视图上的修改也会自动更新到模型上。

第1天，我们会专门讨论依赖注入、服务、资源和自动化测试。第2天，我们会专注在控制器、视图、指令和自动化数据双向绑定上。第3天，会通过过滤器进行数据操作和使用 AngularJS 的路由提供者（route provider）实现浏览器跳转支持。

有这么多激动人心的功能等着我们去探索，那我们就开始吧！

3.2 第1天：使用依赖注入

今天我们会通过了解依赖注入和服务来学习 AngularJS。讨论完这些基础后，我们会创建一个样例应用。在这个样例应用里，我们会定义资源并与服务器端交互，还会写出自动化测试来运行我们的代码。今天会很充实的！

AngularJS 一个最突出的功能就是它的依赖注入容器。如果没有它，组件之间的直接函数调用会一直混在一起；相反，通过定义服务并指明每个服务所需的其他服务，可以将关注点清晰地分离开。

依赖注入在很大程度上保持了代码模块化并易于被测试。每个服务都是一段代码或是一个模块，其中指明了它所有的依赖，但并不关心这些依赖是怎样被创建的，只是被作为函数参数传递进来。一个服务就像是在为客户端扮演着某个角色，客户端并不关心哪个服务来做这些工作，只要这个角色被扮演了就好。

¹ 译者注：指令也可以以 HTML 标签，HTML 标签的 class 属性值或注释的形式存在

依赖注入也十分利于对代码的测试。当你测试一个服务时，想测试的是它的功能，而不是它的依赖。为了能将依赖分离开，只专注在待测的服务上，就可以按照需要来传递模拟（mock）对象完成测试。就我们关注的服务而言，测试其实只是它的一个消费者而已。

让我们先快速地通过代码来体会一下它们的含义。例如你有一个服务 `MyService`，它需要另外一个服务 `MyHelper`，并且会调用这个服务的 `doSomething` 方法。如以下代码所示，`MyService` 负责创建了 `MyHelper` 的实例：

```
angularjs/public/concepts/concepts-test.js
var MyService = function() {
  var myHelper = new MyHelper();
  var result = myHelper.doSomething("test");
  // ...
};
```

这段代码的问题是不易于测试。`MyService` 方法使用的是真正的 `MyHelper` 类的实例，连同 `MyHelper` 自身可能的依赖，这会使测试很难专注在 `MyService` 自己的功能上。

使用依赖注入，`MyService` 不再负责创建 `MyHelper` 的实例。取而代之的是，将 `MyHelper` 的实例作为一个参数传递到 `MyService` 方法中来：

```
angularjs/public/concepts/concepts-test.js
var MyService = function(myHelper) {
  var result = myHelper.doSomething("test");
  // ...
};
```

这使得 `MyService` 更简单，更易于测试。我们可以创建一个单独的 `MyHelper` 实例，并按照测试需要来实现 `doSomething` 方法，然后将这个实例传递进入 `MyService` 方法。在实际应用代码中，传入 `MyService` 的则是真正的 `MyHelper` 类的实例，这是通过 AngularJS 的依赖注入机制来完成的。

现在我们已经了解依赖注入是怎样使代码更模块化和更易于测试了，下面让我们更进一步来创建一个 AngularJS 应用。我们会了解到 AngularJS 这些功能特性是怎样结合在一起的。

3.2.1 你好，AngularJS

依赖注入和自动化数据双向绑定对于任何 AngularJS 应用都是极其重要的，让我

们通过一个简单的例子来了解一下。无论如何，请以此作为起点自己来写些代码尝试一下吧。

打开样例代码文件包中的 `angularjs/public` 文件夹，你会找到一个名为 `index-basic.js` 的文件，文件中有如下代码：

`angularjs/public/index-basic.js`

```

1 var app = angular.module("BasicApp", []);

2 app.service("greeter", function() {
    this.name = "";
    this.greeting = function() {
        return (this.name) ? ("Hello, " + this.name + "!") : "";
    };
});

3 app.controller("BasicController", function($scope, greeter) {
    $scope.greeter = greeter;
});

```

让我们分解来看。

① 对 `angular.module` 的调用创建了 AngularJS 应用。像例子中的 `BasicApp` 一样，你可以为应用命名，并指定需要的插件列表或者一个空列表：[]。

② 然后通过调用 `angular.module` 的返回值（例子中名为 `app` 的变量）的方法来定义服务、控制器等。我们定义了一个名为 `greeter` 的服务，它有一个 `name` 属性和一个名为 `greeting` 的方法，这个方法会根据名称属性的值返回相应的问候语。

③ 最后我们创建了一个名为 `BasicController` 的控制器，它需要 `$scope` 和我们的 `greeter` 服务。`$scope` 是由 AngularJS 提供的，用来为视图提供对象。通常，AngularJS 提供的对象会有 `$` 的前缀来区别于我们自己代码中的对象。现在我们在作用域（`scope`）上定义了 `greeter` 属性为 `greeter` 服务，这样视图就可以访问 `greeter` 了。

下面，我们来看一下视图。打开名为 `index-basic.html` 的文件，文件中有如下代码：

`angularjs/public/index-basic.html`

```

<!doctype html>
1 <html lang="en" ng-app="BasicApp">
  <head>

```



```

    <meta charset="utf-8">
    <title>AngularJS Basic</title>
  </head>
  ② <body ng-controller="BasicController">
    <div>
      What is your name?
      ③ <input type="text" ng-model="greeter.name">
    </div>
    <div>
      ④ {{greeter.greeting()}}
    </div>
  </body>
  ⑤ <script src="lib/angularjs/1.0.8/angular.js"></script>
    <script src="index-basic.js"></script>
  </html>

```

这看上去像是普通的 HTML。使它成为 AngularJS 视图的元素是那些以 ng-开头的属性以及双大括号：{{}}。让我们仔细地看一下每一段。

① <html>元素上的 ng-app 属性开启了这个应用。注意这个属性的值与我们在 JavaScript 代码里调用 angular.module 方法时给应用定义的名称 BasicApp 是一样的。

② 然后，我们在页面元素上使用 ng-controller 属性来表示想使用控制器，在此例中为 BasicController。一个页面可以有多个控制器。我们不用担心控制器之间的冲突，因为每个控制器都是限定在其对应的带有 ng-controller 属性的元素及其子元素内的。

③ 通过控制器的元素和元素的子元素，我们可以使用控制器与 \$scope 相关联的属性，在此例中即 greeter。首先，我们将一个输入框通过 ng-model 属性与 greeter.name 相绑定。

④ 然后，我们在页面通过 {{}} 语法调用 greeter.greeting() 来显示结果。虽然这看起来很像 Mustache 或 Handlebars 的语法，但实际上是由 AngularJS 自己的模板引擎解析的。

⑤ 最后，页面要像加载我们自己的 index-basic.js 代码一样加载 angular.js 脚本。

在浏览器里打开 index-basic.html 文件，你会看到“你的名字是什么？”的文字和一个输入框。输入名称后，问候语会显示出来，如下图所示。

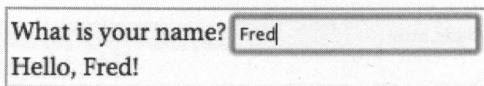


图 3-2 问候语

注意当你输入名称时，问候语会随之变化。这就是 AngularJS 的自动化双向绑定功能在起作用。

虽然这个例子很简单，但我们也完成了不少有意思的 AngularJS 任务。我们定义了服务，使用了依赖注入，创建了控制器，在视图中使用了指令，并利用自动化双向绑定完成了一些功能。这对于第一个应用来说已经很棒了。现在我们来了解一下各部分是怎么工作在一起的，讨论一下每个概念的细节。让我们从服务开始吧！

3.2.2 创建服务

在 AngularJS 应用中，服务会通过代码段向应用的不同部分提供功能，包括控制器、过滤器、其他服务等。通过依赖注入，AngularJS 会将所有服务整合在一起。服务和依赖注入是很搭配的组合，因为服务的代码可以保证应用的模块化，使用依赖注入可以为你节省写实例化服务和连接服务这样的样板代码。

一旦你定义好了一个模块，就可以使用 `service` 方法来创建一个服务，指明服务的名称和包含这个服务的函数。如果这个服务依赖于其他服务，则在函数参数中指明其他服务的名称。下面是一个例子：

```
angularjs/public/concepts/concepts-test.js
var app = angular.module("TestApp", []);
app.service("serviceA", function() {
  this.name = "A";
});
app.service("serviceB", function() {
  this.name = "B";
});
➤ app.service("serviceC", function(serviceA, serviceB) {
  serviceA.name; // returns "A"
  serviceB.name; // returns "B"
});
```

当创建 `serviceC` 时，`serviceA` 和 `serviceB` 会被注入进来，因为参数的名称与服务名称相匹配。

使用字符串名称而不是参数名称

将参数名称与服务名称匹配很简单。这个方法也可以让你添加、删除或重新排列

参数而不用担心代码的同步性，几乎是定义依赖关系最简单的方法。事实上，我们要在样例代码中使用这个方法。

不过压缩代码会使对参数名称的依赖失效，因为代码最小化压缩会简化参数的名称来减少代码的大小。针对这个问题，AngularJS 提供了另外一种策略：用字符串指定服务名称。这就解决了代码最小化压缩带来的问题，因为字符串在压缩后仍然会保持原样。

第一种通过字符串指定服务的方法是在服务函数里定义一个 `$inject` 属性，值为需要注入的服务名称组成的字符串数组：

```
angularjs/public/concepts/concepts-test.js
app.service("serviceA", function() {
  this.name = "A";
});
app.service("serviceB", function() {
  this.name = "B";
});
➤ var svcC = function(svcA, svcB) {
  svcA.name; // returns "A"
  svcB.name; // returns "B"
};
➤ svcC.$inject = ["serviceA", "serviceB"];
app.service("serviceC", svcC);
```

当定义 `svcC` 时，参数名称 `svcA` 和 `svcB` 并不匹配服务的名称 `serviceA` 和 `serviceB`。这种不匹配可能是由最小化压缩或简化变量名称造成的。当创建 `$inject` 属性时，我们已经通过名字指定了依赖关系。服务名称的顺序必须与函数参数的名称顺序一致，因为它们是按照顺序来匹配的。

你现在可以随意改变函数参数名称而不会影响依赖注入了。另外，你需要关注 `$inject` 字符串数组与注入服务名称的一致性，并保证它们在顺序和数量上的匹配。

在之前的代码例子中，你会发现使用 `$inject` 让你通过三步就创建了一个服务：将函数与变量关联，在变量上定义 `$inject` 和调用 `service` 函数来创建服务。

更简洁地使用字符串名称

第二种使用字符串名称指明依赖注入的方式不再需要使用临时变量并添加 `$inject` 属性。取而代之的是，你可以通过仅仅一步来定义服务。当调用 `service` 函数时，指定服务名称及其之后的数组参数，而不是函数参数。在这个数组里，通过字符

串指定了依赖关系和服务函数：

```
angularjs/public/concepts/concepts-test.js
```

```
app.service("serviceA", function() {
  this.name = "A";
});
app.service("serviceB", function() {
  this.name = "B";
});
➤ app.service("serviceC", ["serviceA", "serviceB", function(svcA, svcB) {
  svcA.name; // returns "A"
  svcB.name; // returns "B"
}]);
```

这比 `$inject` 的策略更简单，而且不会受代码最小化压缩导致参数名称变化的影响。但还是需要将服务名称的顺序与函数参数的顺序相匹配。不过这个方法维护起来更简单，因为服务的名称和函数的参数在代码中的位置很接近。

使用 `service` 函数只是创建服务的一种方法，`factory` 函数是另外一种方法。下面我们来看一下两种方法的区别以及如何选择。

使用服务和工厂模式

当我们使用 `service` 的时候，会通过服务函数对 `this` 的属性进行赋值。随后，AngularJS 用我们函数上的 `new` 创建服务。当我们的服务是个对象时，这个方法很好用。`factory` 函数也创建了一个服务，但它直接使用我们返回的结果，而不是调用 `new`。当你希望自己的服务是一个函数或是调用另外一个函数的返回值，而不是通过 `new` 创建一个对象，并添加属性到 `this` 的时候，使用 `factory`。

来看一段简单的例子。如果我们想让一个服务是一个函数，就通过 `factory` 来定义它：

```
angularjs/public/concepts/concepts-test.js
```

```
app.factory("deleteBookmark", function() {
  return function(bookmark) {
    // delete the bookmark...
  };
});
```

通过使用 `deleteBookmark` 服务，我们简单地将它注入到另外一个服务中，并以函数的方式直接调用它：


```
angularjs/public/concepts/concepts-test.js
```

```
app.service("someService", function(deleteBookmark) {  
    var bookmark = ...;  
    deleteBookmark(bookmark);  
});
```

service 和 factory 函数提供了两种方便的方法来创建服务，我们会在样例代码中使用。

现在开始构建我们的应用，然后开始之后的章节。

3.2.3 换个角度来看我们的书签应用前端

通过使用服务、依赖注入和其他 AngularJS 特性，我们会创建如图 3-3 所示的书签应用前端展示。这个应用通过第 1 章第 1 页的 Sinatra 来连接书签服务器，并且与我们在第 2 章第 35 页创建的 CanJS 应用十分相似。同样，我们会与服务器交换数据，负责渲染视图并处理用户交互。我们会构建一个与 CanJS 应用相同的 AngularJS 应用，从而可以很好地比较这两种 JavaScript 框架。

Bookmarking App

Bookmark:

URL: required

Title: required

Tags: (separated by commas)

Filtered by tag: Frameworks | Clear filter

- AngularJS (<http://angularjs.org>) | Frameworks | JavaScript |
- CanJS (<http://canjs.us>) | Frameworks | JavaScript |
- Sinatra (<http://sinatrarb.com>) | Frameworks | Ruby |

Tags:

- Books (1)
- Computer (1)
- Frameworks (3)
- JavaScript (2)
- Ruby (1)

图 3-3 完成的 AngularJS 书签应用

首先，我们会从服务器端获取书签列表的数据。使用 AngularJS 资源服务，我们

会自动处理 Ajax 请求和响应，与服务器端交换数据。这十分简便，因为它很好地减少了我们必须实现的连接客户端到服务器端的代码量。另外，我们也会创建服务来保存和删除书签。最后，我们会以验证代码行为的自动化测试来结束这一天。

3.2.4 使用资源服务

当我们需要一种方法来在客户端和服务器端交换数据时，Ajax 是标准做法。通过 AngularJS 的资源服务，我们就不用在底层的 Ajax 请求和响应上花费精力了。我们可以定义一个资源和 URI。当与向 Sinatra 书签服务器提供的 REST API 进行搭配时，我们可以让 AngularJS 来处理数据交互。

使用资源服务之前，我们先要加载对应的 JavaScript 文件，因为这个服务不是 AngularJS 核心文件的一部分：

```
angularjs/views/index.mustache
```

```
<script src="/lib/angularjs/1.0.8/angular.js"></script>
```

```
➤ <script src="/lib/angularjs/1.0.8/angular-resource.js"></script>
```

我们使用的是 AngularJS 提供的资源服务 \$resource。注意，AngularJS API 一般会使用 \$ 前缀来区分框架代码和应用代码。

```
angularjs/public/app/base/app.js
```

```
angular.module("App_base", ["ngResource"])
```

\$resource 的第一个参数是 URI。我们可以在 URI 中使用以:为前缀的参数，就像在 Sinatra 和 CanJS 里一样。在我们的例子里，URI 含有一个:id 参数。

```
angularjs/public/app/base/app.js
```

```
app.factory("Bookmark", function($resource) {
  return $resource("/bookmarks/:id", {id:"@id"});
});
```

当我们调用资源函数时，会通过引入一个带 id 属性的对象来指定值。如下例所示：

```
var bookmark = Bookmark.get({id:42}); // GET /bookmarks/42
bookmark.title = "changed";
bookmark.$save({id:bookmark.id}); // POST /bookmarks/42
bookmark.$delete({id:bookmark.id}); // DELETE /bookmarks/42
```

当我们从服务器取到书签对象时, `bookmark.id` 的值是 42。当调用 `$save` 和 `$delete` 函数时, 我们用 `{id:bookmark.id}` 来指定 `id` 参数。这个重复是可以被消除的。当调用 `$resource` 函数时, 我们通过第二个参数来指定 `URI` 参数的默认值。例如, `$resource("/bookmarks/:id", {id:1})` 会在 `id` 没有指定时将它设置为 1。当然, 硬编码也不是特别有效。而我们可以用一个以 `@` 为前缀的字符串来告诉 AngularJS 使用调用 `$save` 和 `$delete` 时对象上的 `id` 属性, 如上例可以修改为 `$resource("/bookmarks/:id", {id:"@id"})`。我们的例子变为如下所示:

```
var bookmark = Bookmark.get({id:42}); // GET /bookmarks/42
bookmark.title = "changed";
bookmark.$save(); // POST /bookmarks/42
bookmark.$delete(); // DELETE /bookmarks/42
```

结果虽是一样的, 但我们不再需要在调用 `$save` 和 `$delete` 时指定 `{id:bookmark.id}`。因为我们已将 `{id:"@id"}` 作为 `URI` 的 `id` 参数的默认值, 所以这些都是自动完成的。

我们现在有了一个 `Bookmark` 服务, 它定义了一个资源来与服务器端交换书签数据。我们可以通过调用 `Bookmark` 上的 `query()` 函数来获取书签列表。我们会使用依赖注入来获得 `Bookmark` 服务并且用另外一个名为 `bookmarks` 的服务生成书签列表:

```
angularjs/public/app/base/app.js
app.factory("bookmarks", function(Bookmark) {
  return Bookmark.query();
});
```

现在我们有 `bookmarks` 服务提供的书签列表。虽然这个列表包含可以用来向服务器端发送请求的书签资源对象, 但这个书签 `list` 并不是随着添加删除书签自动更新的。当我们保存一个新书签时, 需要手动将它添加到列表中。删除一个书签后, 我们也不得不将它从列表中移除。但修改已经存在的书签后, 我们就不需要再额外做什么了, 因为这个书签已经在列表里了。

通过使用书签对象的 `$save` 函数, 我们可以将书签保存回服务器端。因为书签是一个资源, 所以调用 `$save` 会自动触发一个 `POST` 请求。为了检查书签是否是新建的, 以决定是否需要将它加到书签列表里, 我们会检查 `id` 的值是否存在:

```
angularjs/public/app/base/app.js
app.factory("saveBookmark", function(bookmarks) {
  return function(bookmark) {
```

```

    if (!bookmark.id) {
      bookmarks.push(bookmark);
    }
    bookmark.$save();
  };
});

```

我们可以通过调用书签上的 `$delete` 函数来删除一个书签，这时会发送一个 DELETE 请求到服务器端。我们也要从标签列表里删除这个标签：

```

angularjs/public/app/base/app.js
app.factory("deleteBookmark", function(bookmarks) {
  return function(bookmark) {
    var index = bookmarks.indexOf(bookmark);
    bookmark.$delete();
    bookmarks.splice(index, 1);
  };
});

```

我们已经创建了服务，以在客户端和服务端创建、读取、更新和删除标签。虽然还没有用户界面，但我们希望能确保现在的代码是工作的。一个好的办法就是写自动化测试。准备好了吗？让我们继续吧。

3.2.5 为服务写自动化测试

AngularJS 社区有很多关于通过自动化测试来验证应用代码是否工作准确的讨论。这个框架提供了极大的测试支持，并且文档用例通常会附上相应的测试。AngularJS 十分重视测试驱动开发，而且理由很充分。通过测试驱动开发（TDD），你可以为自己的代码开发出一套测试，并通过运行测试确保代码的正确性。这能让你放心大胆地重构代码，因为自动化测试会告诉你现在的修改是否正确。

现有的运行 JavaScript 测试的工具很多。在下面的例子中，我们会使用 Jasmine，因为它易搭建，没有依赖，并提供了一套全面直观的 API¹。此外，明天我们会使用 AngularJS 的端到端测试工具，它也遵循着 Jasmine 的语法。今天我们先来写单元测试，从一个 HTML 页面开始运行 Jasmine 吧：

```

angularjs/public/index-test.html
<html lang="en">

```

¹ <http://pivotal.github.io/jasmine>


```

<head>
  <meta charset="utf-8">
  <title>Jasmine Spec Runner</title>
  <link rel="stylesheet" type="text/css" href="lib/jasmine-1.3.1/jasmine.css">
</head>

<body>
</body>

<script src="/lib/jasmine-1.3.1/jasmine.js"></script>
<script src="/lib/jasmine-1.3.1/jasmine-html.js"></script>

<script src="/lib/angularjs/1.0.8/angular.js"></script>
<script src="/lib/angularjs/1.0.8/angular-resource.js"></script>
➤ <script src="/lib/angularjs/1.0.8/angular-mocks.js"></script>

<script src="/app/base/app.js"></script>
➤ <script src="/app/base/app-test.js"></script>

<script src="/test/test-runner.js"></script>
</html>

```

页面会加载 Jasmine 需要的 CSS 和 JavaScript 文件,还有我们的应用和 AngularJS 需要的 JavaScript 文件。特别要注意 angular-mocks.js 和 app-test.js 文件,前者是一个 AngularJS 服务的模拟库;后者是我们要写测试的地方。

我们将从如下结构开始写 Jasmine 测试:

```

angularjs/public/app/base/app-test.js
describe("base/app-test.js", function() {
  beforeEach(function() {
    module("App_base");
  });

  // Add tests here
});

```

describe 和 beforeEach 函数是由 Jasmine 提供的。通过 describe,我们会为一系列测试创建一个容器。正如其名, beforeEach 函数会在每个测试运行前运行。我们会通过 angular-mocks 库提供的 module 函数来加载我们的应用,同时已经准备好添加测试了。在 describe 代码块里,我们可以嵌套添加另外一个 describe 代码块用来测试书签资源。我们会验证是否 bookmarks 服务成功通过 GET 请求从服务器端获取书签列表。为了测试 Ajax 请求并提供一个模拟响应来更好地测试我们的测试用例,AngularJS 模拟库提供了 \$httpBackend 服务:

```
angularjs/public/app/base/app-test.js
```

```
describe("Bookmark resource", function() {
  var mockBookmarks = null;
  ❶ beforeEach(inject(function($httpBackend, Bookmark) {
    mockBookmarks = [
      new Bookmark({id:1}), new Bookmark({id:2}), new Bookmark({id:3})
    ];
    ❷ $httpBackend.expectGET("/bookmarks").respond(mockBookmarks);
  }));
  ❸ it("should retrieve bookmarks", inject(function($httpBackend, bookmarks) {
    $httpBackend.flush();
    ❹ expect(bookmarks.length).toBe(mockBookmarks.length);
  }));

  // add more tests here
});
```

❶ 在调用 `beforeEach` 函数时，我们可以在测试代码中使用 `inject` 函数来从依赖注入中受益。这里，我们注入 `$httpBackend` 服务和我们自己的 `Bookmark` 模型。为了方便测试，我们用后者来创建一个模拟书签列表。

❷ `$httpBackend` 服务提供了一系列方法来指明哪些请求是期望的，以及如何响应它们。因为会使用我们的 `bookmarks` 服务来从服务器端获取书签列表，我们期望有一个发往 `/bookmarks` URI 的 GET 请求。作为响应，我们会返回模拟的书签列表。

❸ Jasmine 库提供的 `it` 函数表示一个测试用例。第一个参数是描述测试用例的字符串。这个 `it` 函数的名字与之后描述的名字拼在一起，读起来会像一个英语句子。

就像 `beforeEach` 一样，我们可以使用 `inject` 函数来注入依赖。在 `it` 函数体中，我们调用 `$httpBackend` 的 `flush()` 来发送等待中的请求。记住 Ajax 请求是异步的，为异步函数调用写测试是一件困难而易出错的事情。为了使其简单，模拟 `$httpBackend` 对象保存着所有等待中的请求让我们可以通过 `flush()` 函数控制什么时候发送它们。这让我们的代码保持线性并易于读懂。

❹ 在 `it` 代码块内，我们通过调用 Jasmine 的 `expect` 方法来指定测试通过的条件。传入实际值后，我们调用另一个称为匹配器的函数，并传入期望值。`toBe` 匹配器会检查两个值是否相同。Jasmine 提供了很多种匹配器，你也可以根据需要自己定义。

从源代码进入 `angularjs` 文件夹，通过运行 `ruby app.rb` 命令来启动 Web 服务器。然后，用浏览器导航到 `http://localhost:4567/index-test.html`。Jasmine 测试运行程序会执行所有测试并生成结果报告，类似图 3-4 所示。

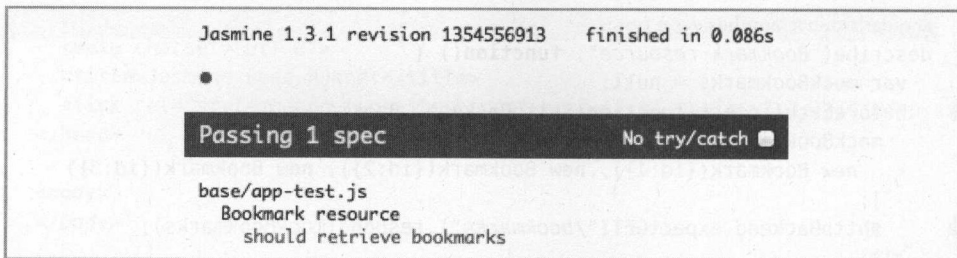


图 3-4 Jasmine 测试运行程序

让我们为保存书签加一个测试：

```
angularjs/public/app/base/app-test.js
it("should save a bookmark", inject(
  function($httpBackend, Bookmark, bookmarks, saveBookmark) {
    $httpBackend.flush();

    $httpBackend.expectPOST("/bookmarks").respond({id:4});
    saveBookmark(new Bookmark({url:"http://angularjs.org", title:"AngularJS"}));

    $httpBackend.flush();
    expect(bookmarks.length).toBe(mockBookmarks.length + 1);
  }
));
```

我们已经为测试用例创建了另一个 it 代码块。通过控制 \$httpBackend 发送请求得到 bookmarks 列表，我们期望一个 POST 请求和一个包含新创建书签 ID 的响应。调用 saveBookmark 会发送 POST 请求。最后通过再次控制 \$httpBackend 发送等待的保存书签请求，我们验证到 bookmarks 列表又多了一个书签。

通过类似的方式，我们可以再为删除书签添加一个测试：

```
angularjs/public/app/base/app-test.js
it("should delete a bookmark", inject(
  function($httpBackend, bookmarks, deleteBookmark) {
    $httpBackend.flush();
    var bookmark = bookmarks[0];

    $httpBackend.expectDELETE("/bookmarks/" + bookmark.id).respond(200);
    deleteBookmark(bookmark);

    $httpBackend.flush();
    expect(bookmarks.length).toBe(mockBookmarks.length - 1);
  }
));
```

这次我们从书签列表里拿到第一个书签，搭建好\$httpBackend来期望一个包含书签ID的DELETE请求。相应的，我们会简单返回一个200 OK HTTP状态码。删除书签后，我们期望bookmarks列表中少一个书签。

更改完代码后，刷新浏览器来重新加载http://localhost:4567/index-test.html并观察结果。

单元测试是极好的验证代码行为的方法。通过Jasmine的语义化API和AngularJS的模拟库，我们能轻松地AngularJS服务撰写和运行单元测试。

3.2.6 我们在第1天学到的

我们在探索AngularJS的第1天发现了AngularJS与众不同的方面，学到了模块、服务、依赖注入、资源和自动化测试。我们已经接触到大部分功能特性了。

第1天的自学

查阅

- AngularJS API 文档。
- 除了service和factory以外的两种定义服务的方法。
- angular-seed项目可以用于你的AngularJS应用模板。

实践

- 使用我们的Sinatra的REST API来写一个服务，使用/bookmarks/:tag链接通过标签提取书签。
- 写一个Jasmine测试来验证你的服务的行为并在浏览器中执行它们。

3.3 第2天：创建控制器和视图

就像MVC架构一样，AngularJS控制器精心地在模型和视图之间提供数据流。我们会在第2天学习控制器和怎样通过指令来动态创建视图。我们会仔细地了

\$scope，它是控制器和视图之间的强力胶。为了实践所学，我们会为书签列表创建一个视图，并为创建和编辑书签创建一个表单。完成后，会看到如图 3-5 所示的前端显示。

Bookmarking App

Bookmark:

URL:

Title:

- AngularJS (<http://angularjs.org>)
- CanJS (<http://canjs.com>)
- Sinatra (<http://sinatrarb.com>)

图 3-5

到目前为止，我们已经从服务器端通过资源获得数据并转成数据对象。今天我们会完成到视图的数据流，将数据注入控制器中，并使视图关于作用域的指令可以使用这些数据，就像图 3-6 从服务器端到视图的数据流中描绘的一样。让我们从学习怎样创建控制器开始这一天吧。

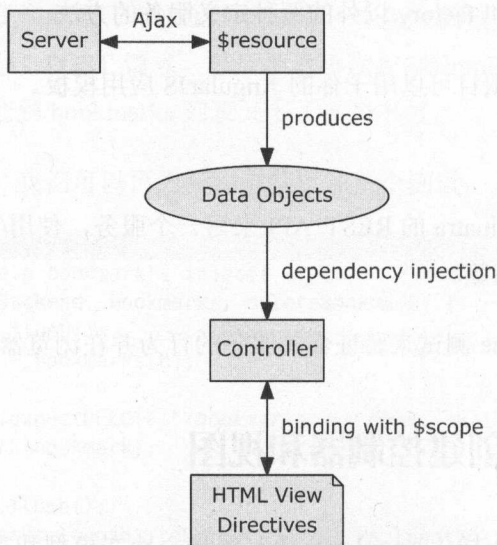


图 3-6 从服务器端到视图的数据流

3.3.1 创建控制器和使用视图指令

现在我们将要为书签列表创建一个控制器和一个视图，这会为我们的应用创建一个用户界面。视图会显示书签以及对应的编辑删除按钮，而控制器会提供回调函数来处理按钮触发的事件。就像 `service` 和 `factory` 函数创建服务一样，`controller` 函数会创建控制器并为其命名，还会有一个函数通过参数名称指明依赖关系：

```
angularjs/public/app/base/app.js
app.controller("BookmarkListController",
  function($scope, bookmarks) {
    $scope.bookmarks = bookmarks;
  }
);
```

通过依赖注入，控制器获得了 `$scope` 和我们的书签列表。AngularJS 提供的 `$scope` 是控制器和视图之间沟通的桥梁。对 `$scope` 的 `bookmarks` 属性赋值后，可以使其对视图可用。

为了能在视图中使用控制器，我们会使用以 `BookmarkListController` 为控制器名称的 `ng-controller` 指令：

```
angularjs/views/base.html
<div
  ng-controller="BookmarkListController"
  ng-include src="'/app/base/bookmark_list.html'">
</div>
```

`ng-controller` 指令通过 `<div>` 元素与控制器相关联。`<div>` 中的所有元素都可以引用控制器赋给 `$scope` 对象的属性。

`ng-include` 指令通过 URI 加载了一个模板。因为指令属性值是代码表达式，所以我们需要将 URI 包含在一个单引号中以得到字符串字面量。这里我们加载了 `bookmark_list.html` 模板。使用 `ng-include` 是一种方式，我们也可以直接将 HTML 模板写入 `<div>` 元素中。使用 `ng-include` 是因为它可以让我们把模板抽取成独立的文件，让代码更易于管理。

`bookmark_list.html` 模板会渲染书签列表，并在每个书签后面渲染“编辑”和“删除”按钮：

```
angularjs/public/app/base/bookmark_list.html
<ul>
  <li class="bookmark" ng-repeat="bookmark in bookmarks">
```

```

<button ng-click="editBookmark(bookmark)">Edit</button>
<button ng-click="deleteBookmark(bookmark)">Delete</button>
<a href="{{bookmark.url}}">{{bookmark.title}}</a>
( <a href="{{bookmark.url}}">{{bookmark.url}}</a> )
</li>
</ul>

```

我们使用了一些有意思的指令:ng-repeat 指令遍历了\$scope 上的 bookmarks 列表。这个指令为每个列表项渲染了元素,并将当前的书签赋值给 bookmark 变量。在元素内有“编辑”和“删除”按钮,每个按钮都通过 ng-click 指令调用一个函数,并将书签作为参数传入(我们很快就会开始编写 editBookmark 和 deleteBookmark 的回调函数)。最后,我们有对应书签 URL 的两个链接,一个显示为书签的名称;另一个显示为书签的 URL。注意,我们通过使用{{}}语法引用 bookmark 对象的方式。

图 3-7 总结了我们是怎样将 JavaScript 代码中的控制器和作用域关联到 HTML 视图的指令上的。

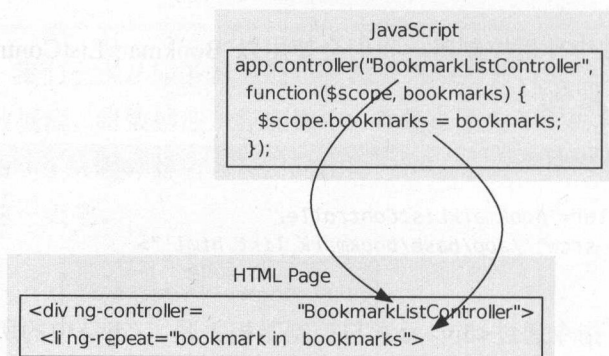


图 3-7 控制器和视图之间的关联

当用户点击书签后面的“编辑”或“删除”按钮时,ng-click 指令会调用 editBookmark 和 deleteBookmark 函数。为了让这两个函数可以被视图调用,我们需要在控制器里注入它们并赋值到作用域上:

```

angularjs/public/app/base/app.js
app.controller("BookmarkListController",
function($scope, bookmarks, deleteBookmark, editBookmark) {
  $scope.bookmarks = bookmarks;
  $scope.deleteBookmark = deleteBookmark;
  $scope.editBookmark = editBookmark;
}
);

```

deleteBookmark 服务是我们之前创建的。editBookmark 服务还不存在，现在我们就来创建。

3.3.2 利用数据双向绑定的优势

当用户点击书签后面的“编辑”按钮时，我们希望在书签表单里编辑对应的书签。当用户保存书签时，我们希望将修改保存回服务器端并更新书签列表。最后，当表单为空白的或用户清除了表单时，我们希望创建一个新的书签。让我们看看如何利用 AngularJS 的数据双向绑定来轻松地完成这一切。

我们会从创建一个名为 state 的服务开始，在这个服务中定义一个供其他服务绑定的属性。可以将这个对象和各个服务的关系想象成作用域对于控制器和视图那样：

```
angularjs/public/app/base/app.js
app.service("state", function(Bookmark) {
  this.formBookmark = {bookmark:new Bookmark()};
  this.clearForm = function() {
    this.formBookmark.bookmark = new Bookmark();
  };
});
```

我们已经添加了 formBookmark 属性，并将其初始化为带一个空书签的对象，还添加了 clearForm 函数用来将 formBookmark.bookmark 重置为一个空书签。这会为书签表单提供方便。

现在我们有了 state 对象，就可以开始写 editBookmark 服务了：

```
angularjs/public/app/base/app.js
app.factory("editBookmark", function(state) {
  return function(bookmark) {
    state.formBookmark.bookmark = bookmark;
  };
});
```

编辑书签其实就是设置待编辑书签的 state 对象上的 formBookmark.bookmark 属性。当我们创建书签表单时，只需要将表单的输入框绑定到 state 对象的 formBookmark 属性上，然后动态数据双向绑定会负责前后修改的同步。让我们

开始吧！

3.3.3 创建书签表单

有了数据双向绑定，在 AngularJS 中创建表单就很容易了。像其他视图模板一样，表单也是普通的 HTML 加上 AngularJS 指令。ng-model 指令将一个输入框绑定到一个模型属性上，而<form>元素上的 ng-submit 指令会在用户提交表单时调用一个函数。

让我们在 bookmark_form.html 文件中搭建用来创建编辑书签的表单吧。表单的模板如下所示：

```
angularjs/public/app/base/bookmark_form.html
Bookmark:
<form ng-submit="saveBookmark(formBookmark.bookmark)">

  <label>
    URL:
    <input type="text" ng-model="formBookmark.bookmark.url" name="url">
  </label>

  <label>
    Title:
    <input type="text" ng-model="formBookmark.bookmark.title" name="title">
  </label>

  <input type="submit" class="btn btn-primary" value="Save">
  <input type="button" class="btn" ng-click="clearForm()" value="Clear">

</form>
```

通过使用 ng-model 指令，当输入框里的值改变时，AngularJS 会自动更新模型上的值，反之亦然。因此，我们不用再写任何代码来双向传递表单的值。

想保存书签，我们只需在<form>元素上添加 ng-submit 指令，它会调用控制器上为书签表单准备的 saveBookmark 函数。我们还会通过清空表单按钮上的 ng-click 指令调用控制器的 clearForm 函数。

让我们继续完成书签表单控制器吧。我们需要将 formBookmark、saveBookmark 和 clearForm 属性设置到\$scope 上。

```
angularjs/public/app/base/app.js
app.controller("BookmarkFormController",
  function($scope, state, bookmarks, saveBookmark) {
    $scope.formBookmark = state.formBookmark;
    $scope.saveBookmark = saveBookmark;
    $scope.clearForm = state.clearForm;
  }
);
```

我们已经将作用域的 `formBookmark` 和 `clearForm` 属性绑定到 `state` 对象的相应属性上了。有趣的是，当我们改变 `state` 对象上的 `formBook.bookmark` 的值时，作用域会自动将修改同步到视图的表单输入框上。这样编辑书签就生效了，因为 `editBookmark` 函数会编辑同一个状态对象上的 `formBookmark.bookmark` 属性来设置待编辑书签。

还记得吗，我们已经创建了一个 `saveBookmark` 服务来添加新书签到 `bookmarks` 列表中。现在当通过 `$save` 保存这个书签到服务器端后，我们还会清空表单：

```
angularjs/public/app/base/app.js
➤ app.factory("saveBookmark", function(bookmarks, state) {
  return function(bookmark) {
    if (!bookmark.id) {
      bookmarks.push(bookmark);
    }
    bookmark.$save();
    state.clearForm();
  };
});
```

现在我们准备好 `BookmarkFormController` 需要的所有部分，可以用到视图上了：

```
angularjs/views/base.html
<div
  ng-controller="BookmarkFormController"
  ng-include src="'/app/base/bookmark_form.html'">
</div>
```

太棒了！双向动态绑定通过作用域和状态对象让所有值保持同步。我们的代码还保持着松耦合性：例如点击书签列表的“编辑”按钮可以在表单中设置书签，但书签列表没有直接绑定到表单上。如图 3-8 书签表单控制器所示，我们的状态对象就像一个中介保持了组件之间的独立。

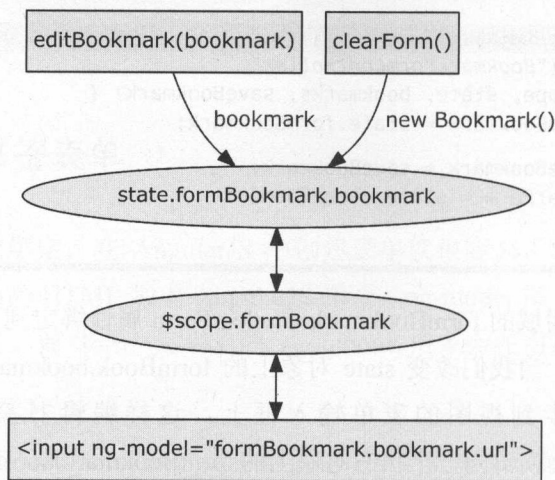


图 3-8 书签表单控制器

在一个没有动态绑定的框架中，书签列表需要包含生成书签表单的代码来编辑选中的书签，并需要更多的代码从表单同步数据到书签对象来保存书签。通过状态对象和动态数据双向绑定，我们不再需要这些同步代码。

3.3.4 关于作用域的重要注意事项

很有必要仔细了解一下我们是怎样在状态对象和作用域上定义 `formBookmark` 属性的，因为这里有一个 AngularJS 开发人员经常会遇到的陷阱。

你可能感觉很奇怪，为什么我们要把有 `bookmark` 属性的对象赋值到 `formBookmark` 变量上，以致不得不通过 `formBookmark.bookmark` 来引用书签？为什么不直接把书签赋值给 `formBookmark` 呢？

如图 3-9 所示，在这个场景下，我们已经将一个空书签赋值给 `formBookmark.state` 对象和 `$scope` 都指向同一个对象。当编辑已存在的书签时，我们将书签复制给 `state` 对象的 `formBookmark` 属性，希望 `$scope` 也可以获得相同的改动。但事与愿违，因为通过重新对 `formBookmark` 赋值，我们已经将引用改成了不同的对象。就像在图最后看到的一样，修改后即破坏了 `state` 和 `$scope` 之间的关联，它们的 `formBookmark` 属性现在已经指向了两个不同的对象。

因为失去了这份关联，在代码中对 `state.formBookmark` 的修改对控制器的

`$scope.formBookmark` 是不可见的。在我们的应用中，这会破坏“编辑”按钮的功能：编辑书签不再会生成书签表单。

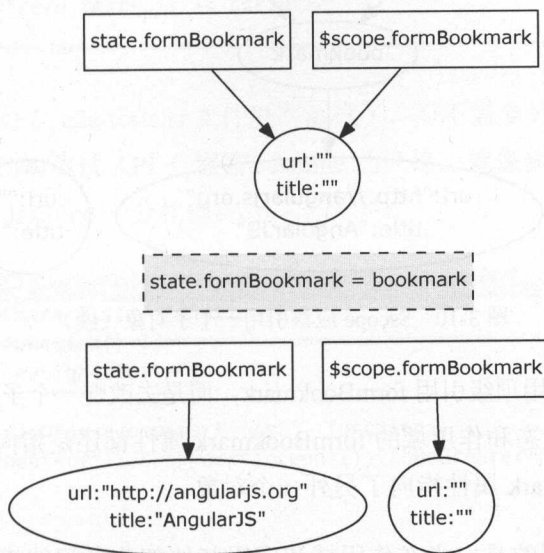


图 3-9 `$scope` 不应该引用根级对象

现在看一下图 3-10 `$scope` 应该引用一个子对象。通过让 `formBookmark` 指向带 `bookmark` 属性的对象并让该属性指向书签，我们可以改变 `formBookmark.bookmark` 引用并始终保持 `state` 和 `$scope` 之间的同步。

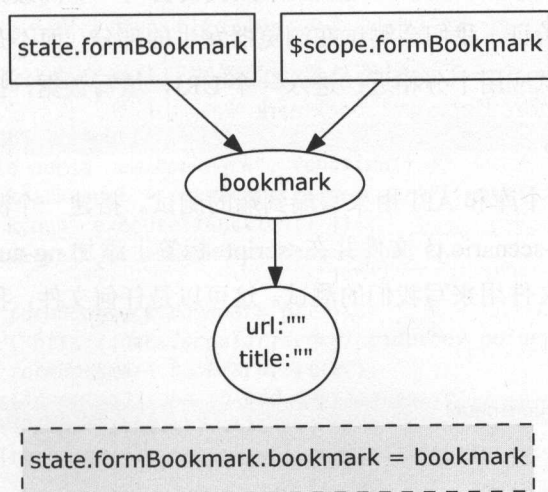


图 3-10 `$scope` 应该引用一个子对象

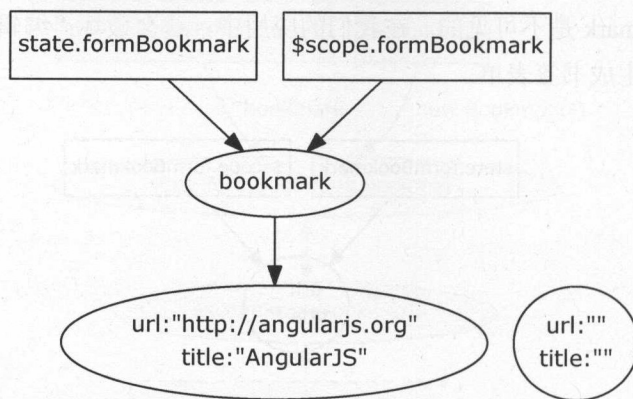


图 3-10 \$scope 应该引用一个子对象 (续)

关键就是不要使用顶级引用 `formBookmark`，而是去改变一个子对象 `formBookmark.bookmark`。这样，状态和作用域的 `formBookmark` 属性都还是指向同一个对象。现在是一个对象的 `bookmark` 属性指向了另外一个对象。

从这里我们学到的是，当在作用域和应用其他部分之间的模型对象中分享引用时，表达式至少应该含有一个点 (.)。

3.3.5 端到端的自动化测试

昨天我们用 `Jasmine` 写了一些单元测试，今天我们会写一些高级别的测试让应用在端到端的环境下得到验证。我们会测试在浏览器发生的部分，而不是独立的代码单元。这类测试与手动测试应用十分相近：进入一个 URI，填写数据，提交表单，然后观察浏览器显示的结果。

AngularJS 提供了一个库和 API 用来写端到端的测试。搭建一个测试运行器，我们仅仅需要加载 `angular-scenario.js` 文件并在 `<script>` 标签上添加 `ng-autotest` 指令，同时我们还需要加载一个文件用来写我们的测试。这可以是任何文件，我们暂且命名为 `e2e-tests.js`：

```
angularjs/public/e2e-test-runner.html
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>AngularJS End-to-End Test Runner</title>
  </head>
```

```

<body>
</body>
➤ <script src="/lib/angularjs/1.0.8/angular-scenario.js" ng-autotest></script>
➤ <script src="/e2e-tests.js"></script>
</html>

```

我们已经准备好在 `e2e-tests.js` 文件里写测试了，而不需要另外学习新的语法，因为 AngularJS 的端到端测试 API 会遵循 Jasmine 的风格。就像你在这里看到的一样，搭建端到端测试与 Jasmine 十分相似：

```

angularjs/public/e2e-tests.js
describe("Bookmark list", function() {
  beforeEach(function() {
➤    browser().navigateTo("/");
  });
  it("should display a bookmark list", function() {
➤    expect(repeater("li.bookmark").count()).toBeGreaterThan(0);
  });
});

```

我们有和 Jasmine 单元测试相同的 `describe`、`beforeEach` 和 `it` 函数。端到端测试不同的是，我们用的是 AngularJS 的场景库来调用函数，比如调用 `browser` 可以发送请求，调用 `repeater` 可以观察网页的内容。这是比我们写的单元测试更高级的测试。这里我们期望页面会显示一个书签列表，也就意味着页面至少要有一个 `bookmark` 类对应的 `` 元素。

让我们写一个测试来通过填写提交表单创建书签。这个测试会检查列表中原有多少书签，然后确认提交表单后书签的数量是否增加了一个：

```

angularjs/public/e2e-tests.js
it("should add a new bookmark", function() {
➊  var bookmarkCount = repeater("li.bookmark").count();
    bookmarkCount.execute(function() {});
    var previousCount = bookmarkCount.value;

➋  input("formBookmark.bookmark.url").
    enter("http://docs.angularjs.org/guide/dev_guide.e2e-testing");
    input("formBookmark.bookmark.title").
    enter("AngularJS end-to-end testing guide");
➌  element("input:submit").click();
➍  expect(repeater("li.bookmark").count()).toBe(previousCount + 1);
});

```

➊ 我们希望在测试开始的时候计算一下书签的数量，之后会用这个值与提交创

建书签表单后的书签数量值作比较。现在 `repeater.count` 没有直接返回一个值，返回的是称作将来的部分。我们需要调用 `execute` 来得到这个值。更重要的是，我们需要提供一个回调函数，否则测试运行器会跳过这个测试。这也是我们为什么要传入一个空函数来 `execute`。

② 通过 `input` 和 `enter` 函数，我们可以模拟在输入框输入值。我们写的代码填入了书签的 URL 和标题值。

③ `element` 函数使用一个 jQuery 类型的选择器寻找元素。找到表单提交按钮后，我们调用 `click()` 来模拟按钮点击。

④ 这里的 `expect(...).toBe(...)` 语法和 Jasmine 是一样的。在这个例子里，我们不需要调用 `repeater.count` 的 `execute` 函数，`expect` 函数会完成一切的。我们计算列表中标签的数量，并确定多了一个书签。如果测试通过，我们就确定了创建新书签的一个基本场景。

通过 `ruby app.rb` 命令来启动服务器，打开浏览器导航至 `http://localhost:4567/e2e-test-runner.html` 就可以运行这个测试了，并会看到如图 3-11 所示的测试结果。

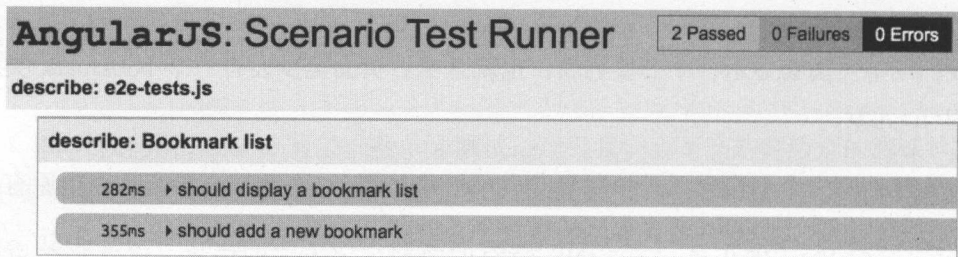


图 3-11 AngularJS 端到端测试运行器

我们已经写了一个填写提交表单并在页面上验证结果的测试。通过单元测试和端到端测试双管齐下，我们用自动化测试很好地武装了我们的 AngularJS 应用，确保了应用的健康性。

3.3.6 我们在第 2 天学到的

在第 2 天，我们从服务进入了控制器和视图的学习。我们讨论了怎样将控制器、

作用域和视图指令组合在一起开发用户界面；创建了书签列表的视图和用来创建编辑书签的表单；利用动态数据双向绑定的优点来保证表单、书签列表和服务器端之间的同步。最后，通过写端到端自动化测试又了解了 AngularJS 对测试的强大支持。确实是高效的一天啊！

第2天的自学

查阅

- AngularJS 指令列表。
- AngularJS 端到端测试库 API。

实践

- 试着使用不同的视图指令并在浏览器验证结果，例如使用 `ng-mouseenter` 和 `ng-mouseleave` 在移动鼠标时显示隐藏书签的标签。
- 写端到端测试来验证视图指令的行为。
- 使用 AngularJS 团队开发的测试运行器 Karma 来运行你的测试。

3.4 第3天：创建过滤器和路由

今天我们会了解更多 AngularJS 特有的数据绑定方式的优势，以此来结束对 AngularJS 的探索。我们会包装我们的书签应用，给它加上标签、过滤器和路由。通过数据绑定，我们不但可以保持组件间清晰解耦，而且还能非常容易地做到这一点。我们的代码主要会关注在模型上，AngularJS 会负责更新视图。

今天，我们第一件要做的事情是通过对书签添加标签列表支持来改进我们的应用。这会涉及添加一个输入框让用户输入书签的标签（以逗号分隔），并将其通过逗号作为分隔符提取出数组。我们还会在书签列表的每个书签后面显示标签，最后会为所有书签构造出一个统一的标签列表。图 3-12 展示了我们期望达到的结果。

Bookmarking App

Bookmark:
 URL:
 Title:
 Tags: (separated by commas)
 Save Clear

All bookmarks

- Edit Delete AngularJS (http://angularjs.org) | Frameworks | JavaScript |
- Edit Delete CanJS (http://canjs.com) | Frameworks | JavaScript |
- Edit Delete Sinatra (http://sinatrarb.com) | Frameworks | Ruby |

Tags:

- Frameworks (3)
- JavaScript (2)
- Ruby (1)

图 3-12 为书签添加标签

3.4.1 为书签添加标签

我们会从为书签表单添加输入框，让用户可以输入标签开始：

```
angularjs/public/app/tagfilter/bookmark_form.html
```

```
<label>
  Tags: (separated by commas)
  <input type="text" ng-model="formBookmark.bookmark.tagList" ng-list
    name="tagList">
</label>
```

我们已经将输入框与书签的 `tagList` 属性绑定在一起了。注意 `<input>` 标签上的 `ng-list` 指令：它会将用户输入的以逗号分隔的字符串转换为一个字符串数组对应到 `tagList` 属性上。例如，如果用户输入 `Frameworks, JavaScript`，`tagList` 属性会包含 `["Frameworks", "JavaScript"]`。

现在保存书签会将 `tagList` 属性也发送回服务器端，所以书签的标签也被保存了。当我们从服务器获取书签数据时，`tagList` 也会被发送；当编辑已存在的书签时，我们需要将 `tagList` 属性转换回以逗号分隔的字符串并填到输入框中。结果表明，`ng-list` 指令已经帮我们完成这些工作了！

我们已经为书签添加了标签支持。现在我们在图 3-12 为书签添加标签所示的屏幕右方显示统一的标签列表，并在每个标签后显示相关书签的数量。

3.4.2 构建一个标签列表

构建标签列表需要遍历所有书签，记录标签和书签的数量。我们会从一个服务开始，这个服务会返回通过书签列表构建标签列表的函数：

```
angularjs/public/app/tagfilter/app.js
app.factory("buildTagList", function() {
  return function(bookmarks) {
  };
});
```

我们可以通过调用这个函数 `buildTagList(bookmarks)` 得到一个包含名称和书签数量的标签列表。

通过这个函数，我们会创建一个包含每个标签的相关书签数量的对象 `bookmarkCounts`：

```
angularjs/public/app/tagfilter/app.js
var bookmarkCounts = {};

bookmarks.each(function(bookmark) {
  var tagList = bookmark.tagList;

  tagList.each(function(tag) {
    var existing = bookmarkCounts[tag];
    bookmarkCounts[tag] = existing ? existing + 1 : 1;
  });
});
```

一旦我们遍历完标签列表，`bookmarkCounts` 对象的键值就是标签的名称。我们会对名称排序，并返回一个包含名称和对应书签数量的对象列表：

```
angularjs/public/app/tagfilter/app.js
var labels = Object.keys(bookmarkCounts);
labels.sort();
return labels.map(function(label) {
  return {label:label, bookmarkCount:bookmarkCounts[label]};
});
```

我们的标签列表准备好了。为了在视图中显示它，我们需要一个控制器来在作用

域上设置标签列表。同时,我们希望标签列表能随着书签列表的改变而更新。注意我们得手动管理书签列表,它不是自动通过动态绑定更新的。想监听一个属性的变化,可以使用\$scope上的\$watch函数,参数为属性名称字符串和一个用来接收更新值的回调函数。在标签列表控制器里,我们通过调用 buildTagList 服务来监听 bookmarks 属性并更新作用域上的 tags:

```
angularjs/public/app/tagfilter/app.js
app.controller("TagListController",
function($scope, state, bookmarks, buildTagList) {
    $scope.bookmarks = bookmarks;
    $scope.$watch("bookmarks", function(updatedBookmarks) {
        $scope.tags = buildTagList(updatedBookmarks);
    }, true); // true compares objects for equality rather than by reference.
});
```

现在我们已经展示出每个标签的名称和对应的书签数量。恭喜大家,已经完成了今天的第一个目标!

```
angularjs/public/app/tagfilter/tag_list.html
Tags:
<ul>
> <li ng-repeat="tag in tags">
    <a href="#"
      ng-click="filterBy(tag)">{{tag.label}} ({{tag.bookmarkCount}})</a>
    </li>
</ul>
```

注意,我们已经让每个标签都是可点击的。当用户点击标签时,我们希望能过滤书签列表,只显示包含当前选中标签的书签列表。这是我们的下一个任务。

3.4.3 通过过滤器操作数据

标签上的链接会调用\$scope上的 filterBy(tag)函数。我们希望这个函数通过标签来过滤书签列表,但也希望书签后面的标签同样有这个功能。现在我们知道完成这个功能的最好办法就是利用数据绑定的优势,以保持代码的简单和组件的松耦合性。现在已经有一个状态对象扮演着我们应用里不同部分之间桥梁的角色,所以让我们为书

签过滤器添加一个属性吧：

```
angularjs/public/app/tagfilter/app.js
```

```
app.service("state", function(Bookmark) {
  this.formBookmark = {bookmark:new Bookmark()};
  this.clearForm = function() {
    this.formBookmark.bookmark = new Bookmark();
  };
  this.bookmarkFilter = {filterTag:""};
});
```

```
angularjs/public/app/tagfilter/app.js
```

```
app.controller("TagListController",
  function($scope, state, bookmarks, buildTagList) {
    $scope.bookmarks = bookmarks;
    $scope.$watch("bookmarks", function(updatedBookmarks) {
      $scope.tags = buildTagList(updatedBookmarks);
    }, true); // true compares objects for equality rather than by reference.

    $scope.filterBy = function(tag) {
      state.bookmarkFilter.filterTag = tag.label;
    };
  }
);
```

就像你预期的那样，过滤标签列表需要绑定到状态对象的这个属性上并过滤 bookmarks 列表。为了完成这个任务，我们要使用 AngularJS 兵器库里另一个武器：过滤器。就像服务和控制器一样，我们只是需要在 filter 函数参数里指定一个名称和一个包含其依赖关系的函数：

```
angularjs/public/app/tagfilter/app.js
```

```
app.filter("filterByTag", function() { // no dependencies
});
```

在这个函数中，我们需要返回另外一个函数，它会对传入的书签数组应用过滤器，并返回一个过滤后的书签数组。这个函数能接受任何其他需要的参数。在我们的例子里，过滤器需要知道过滤书签的标签是什么：

```
angularjs/public/app/tagfilter/app.js
```

```
return function(bookmarks, filterTag) {
  return bookmarks.filter(byTag(filterTag));
};
```

这个函数通过标签过滤书签。bookmarks.filter 函数需要的是一个通过单个书签返回 true 或 false 的函数来指明这个书签是否应该包含在这个过滤后的列表中。这就是

byTag 函数所提供的:

```
angularjs/public/app/tagfilter/app.js
var byTag = function(filterTag) {
  return function(bookmark) {
    var tagList = bookmark.tagList;
    var noFilter = (!filterTag) || (filterTag.length == 0);
    var tagListContainsFilterTag = tagList &&
      tagList.indexOf(filterTag) > -1;
    return noFilter || tagListContainsFilterTag;
  };
};
```

要决定是否留下这个书签,函数会在书签的标签列表里查找过滤的标签。如果待过滤标签不存在或为空,就不需要过滤,书签会被留下;否则,书签的标签列表必须包含这个待过滤标签,书签才能被留下。

在视图中使用过滤器

现在,我们的 filterByTag 过滤器已经完成了。想在视图模板里使用这个过滤器,就需要通过 “|” 字符来使用过滤器,如下代码所示:

```
angularjs/public/app/tagfilter/bookmark_list.html
<li class="bookmark"
  ng-repeat="bookmark in bookmarks | filterByTag:bookmarkFilter.filterTag">
```

在过滤书签列表的时候,我们通过 filterByTag 函数并在其后添加:和待过滤标签名称 bookmarkFilter.filterTag 来指定一个附加的参数。AngularJS 会负责应用我们的过滤器并在视图中渲染出过滤后的书签列表。甚至因为作用域的 bookmarkFilter.filterTag 是绑定在我们的状态对象上的,只要应用里改变了状态对象的 bookmarkFilter.filterTag,AngularJS 就会随之更新过滤后的列表。

为了将标签列表中选择的待过滤标签关联到书签列表上,我们获取状态对象上的 bookmarkFilter,并通过绑定到书签列表控制器的 \$scope 上让视图可以使用它:

```
angularjs/public/app/tagfilter/app.js
app.controller("BookmarkListController",
  function($scope, state, bookmarks, editBookmark, deleteBookmark) {
    $scope.bookmarks = bookmarks;
    $scope.bookmarkFilter = state.bookmarkFilter;
    $scope.filterBy = function(tag) {
      state.bookmarkFilter.filterTag = tag;
    };
  });
```

```

    // ...
  }
);

```

我们还使 `filterBy` 回调函数是可用的。这并不奇怪，因为可以改变状态对象上的 `bookmarkFilter` 对象。现在当渲染每个书签后面的标签列表时，我们也可以通过点击它们链接到过滤后的书签列表了：

```

angularjs/public/app/tagfilter/bookmark_list.html
<span ng-repeat="tag in bookmark.tagList">
  <a href="#" ng-click="filterBy(tag)">{{tag}}</a> |
</span>

```

我们的过滤器现在对于视图完全可用了。

清除过滤器

这里还有一个细节要强调一下。当然，用户现在可以通过点击标签来过滤书签列表了，但他们也需要清除之前的过滤让书签的完整列表可以再次显示出来。这只需要通过设置状态对象上的 `bookmarkFilter.filterTag` 为空字符串即可。让我们在标签过滤控制器里来实现：

```

angularjs/public/app/tagfilter/app.js
app.controller("TagFilterController", function($scope, state) {
  $scope.bookmarkFilter = state.bookmarkFilter;

  $scope.clearFilter = function() {
    state.bookmarkFilter.filterTag = "";
  };
});

```

在这个视图里，不仅将显示当前列表正在被过滤的标签，还会显示一个清除过滤的链接，或当没有过滤的时候直接显示“全部书签”。我们可以使用 `ng-show` 指令来通过一个条件显示隐藏视图模板的某一部分：

```

angularjs/public/app/tagfilter/tag_filter.html
<h3 ng-show="bookmarkFilter.filterTag">
  Filtered by tag: {{bookmarkFilter.filterTag}}
  | <a href="#" ng-click="clearFilter()">Clear filter</a>
</h3>
<h3 ng-show="!bookmarkFilter.filterTag">All bookmarks</h3>

```

我们已经完成任务了，继续来使用一下吧。从 `angularjs` 目录下使用 `ruby app.rb`

来运行服务器，并在浏览器里打开 <http://localhost:4567/example/tagfilter> 链接。添加一些书签及其标签，编辑标签，过滤列表，删除书签，注意页面是怎样相应变化的。这些都归功于 AngularJS 的双向动态数据绑定。

3.4.4 定义路由

像在 CanJS 里那样，我们想让 AngularJS 应用支持浏览器前进后退按钮和书签导航。前端框架通常不会包含这类问题的解决方案，但 AngularJS 通过路由提供者让它成为了可能。

通过路由提供者，我们可以用链接将控制器和模板关联起来。路由链接可以含有参数，控制器可以轻松访问这些参数的值。让我们用这些功能来实现不同标签的过滤和显示全部书签之间的跳转吧。

使用路由提供者，我们需要为 `angular.module` 函数调用添加第三个参数。记住第一个参数是模块的名称，第二个是依赖关系列表，第三个是一个函数，会在创建模块时被调用。这个函数可以使用依赖注入，所以我们可以通过 AngularJS 提供的服务 `$routeProvider` 来定义路由：

```
angularjs/public/app/routing/app.js
angular.module("App_routing", ["ngResource", "App_tagfilter"],
function($routeProvider) {
  var params = {
    controller: "BookmarkListController",
    templateUrl: "/app/routing/bookmark_list.html"
  };
  $routeProvider.
    when("/", params).
    when("/filter/:tag", params);
})
)
```

我们已经将默认链接和 `/filter/:tag` 链接关联到 `BookmarkListController` 上了，也在路由提供者上指定了模板。我们不再需要视图中的 `ng-include src="/app/tagfilter/bookmark_list.html"` 了。现在使用 `ng-view=:`

```
angularjs/views/routing.html
<div
  ng-controller="BookmarkListController"
  ng-view>
</div>
```

当一个路由匹配在路由提供者里的定义时，AngularJS 会在含有 `ng-view` 指令的元素上渲染相应的模板。

为了使用路由，控制器会通过依赖注入获取到 `$routeParams` 对象，其包含匹配到链接的参数属性。例如，当用户访问匹配的 `/filter/:tag` 链接时，`$routeParams.tag` 的值就是链接中 `:tag` 的位置对应的值。我们可以设置状态对象上书签过滤标签的值，像这样：

```
angularjs/public/app/routing/app.js
app.controller("BookmarkListController",
  function($scope, $routeParams, state,
    bookmarks, editBookmark, deleteBookmark) {
    $scope.bookmarks = bookmarks;
    $scope.bookmarkFilter = state.bookmarkFilter;
    state.bookmarkFilter.filterTag = $routeParams.tag;
    $scope.editBookmark = editBookmark;
    $scope.deleteBookmark = deleteBookmark;
  }
);
```

`TagListController` 也是一样：

```
angularjs/public/app/routing/app.js
app.controller("TagListController",
  function($scope, $routeParams, state, bookmarks, buildTagList) {
    $scope.bookmarks = bookmarks;
    state.bookmarkFilter.filterTag = $routeParams.tag;
    $scope.$watch("bookmarks", function(updatedBookmarks) {
      $scope.tags = buildTagList(updatedBookmarks);
    }, true);
  }
);
```

`TagFilterController` 也是一样：

```
angularjs/public/app/routing/app.js
app.controller("TagFilterController",
  function($scope, $routeParams, state) {
    $scope.bookmarkFilter = state.bookmarkFilter;
    state.bookmarkFilter.filterTag = $routeParams.tag;
  }
);
```

像之前一样，一切都通过状态对象生效。这次，我们将 `bookmarkFilter.filterTag`

绑定为路由的标签值。当链接变化时，不管是通过点击链接还是在浏览器地址栏输入链接，或使用书签，标签的值都会被设置到状态对象上，页面也会相应地自动更新。

现在不用在书签列表的标签链接上加 `ng-click="filterBy(tag)"` 了，我们可以直接链接到 `#/filter/{{tag}}`：

```
angularjs/public/app/routing/bookmark_list.html
<span ng-repeat="tag in bookmark.tagList">
➤   <a href="#/filter/{{tag}}">{{tag}}</a> |
</span>
```

我们在标签列表上也做了类似的修改：

```
angularjs/public/app/routing/tag_list.html
Tags:
<ul>
  <li ng-repeat="tag in tags">
➤   <a href="#/filter/{{tag.label}}">{{tag.label}} ({{tag.bookmarkCount}})</a>
  </li>
</ul>
```

想清除过滤，只需要将链接指向#：

```
angularjs/public/app/routing/tag_filter.html
<h3 ng-show="bookmarkFilter.filterTag">
  Filtered by tag: {{bookmarkFilter.filterTag}}
➤   | <a href="#">Clear filter</a>
</h3>
<h3 ng-show="!bookmarkFilter.filterTag">All bookmarks</h3>
```

这些修改不仅会使链接更简洁，也会减少控制器的代码量。我们可以从 `BookmarkListController` 和 `TagListController` 移除 `filterBy` 函数，从 `TagFilterController` 移除 `clearFilter` 函数。可以说，依赖注入和动态数据绑定使我们的任务变成了一件轻松的事。

3.4.5 我们在第3天学到的

学习 AngularJS 的最后一天，我们利用数据绑定的优点构建了强大的功能，专注在模型和控制器之间的关联上，让 AngularJS 来负责视图的渲染。

第3天的自学

查阅

- AngularJS 提供的内置过滤器。
- angular-ui 和 angular-utils 项目。

实践

- 试用 AngularJS 过滤器，如 `limitTo`、`lowercase` 和 `orderBy`。
- 写一个自定义的过滤器来得到没有标签的书签列表。
- 试用 angular-ui 项目提供的 `ng-grid` 组件。

3.4.6 对 AngularJS 创建者 Miško Hevery 的采访

我们：是什么让你想到创建 AngularJS？目前现有的框架缺少的是什么？

Miško：有很多原因一起激励了我实现 Angular。

我意识到开发网络应用就是一个大的封装处理问题，即怎样从数据库获取数据并发送回去。构建了很多网络应用后，我就不想再用同样的方式构建了。

我希望通过添加一些标记为静态网页添加活力，这也解释了指令为什么存在。指令是 Angular 十分核心的功能，也是 JS 框架里很独特的功能。我希望仅仅通过添加一些额外标记就让 `<form>` 充满活力，而不用必须去写很多代码。

我想学习 JavaScript，并构建一个适合初始 JS 项目的框架。除了 JQuery，我不知道任何其他的 JS 框架。

最开始的 Angular 是云上的一个服务，可以让你仅仅用标记就构建十分简单的 CRUD 应用，数据会存在作为服务的云数据库上。后来 Angular 移除了作为服务的数据库，成为了一个通用的网络框架，而不是一个纯 HTML 的声明扩展。

我们：你认为 AngularJS 最好的功能是什么？是什么让 AngularJS 这么与众不同？

Misko: 三点让 Angular 与众不同。

可以通过指令扩展 HTML 的语法库。扩展 HTML 的这个想法很强大, 它可以让开发人员以优雅的声明方式实现代码的目标, 而不是传统的步骤。其他框架也有在 HTML 嵌入声明式信息的, 但语法是固定的。Angular 允许声明的语法被扩展, 这样开发人员就可以将 HTML 转化成领域特有语言或 DSL。

依赖注入在服务器端很常见, 但 Angular 首先在前端引入了这个功能。Angular 是第一个充分发挥依赖注入优点的框架, 效果就是 Angular 应用不需要 main 方法或其他相应的代码来组装应用。

Angular 被设计时就考虑了可测试性。我们 (Angular 团队) 是 TDD 的忠实拥护者, 通过 TDD 构建出 Angular, 并且希望开发人员也能用 TDD 来构建他们的应用。Angular 的很多设计决定都是测试驱动的, 自带端到端测试运行器和为单元测试准备的模拟方式。

我们: 你认为 AngularJS 的前景是怎样的? 你最期待的是什么?

Misko: Angular 已经成为了设计思想的试验台, 且很多已经成为了标准, 比如 Object.observe 和 MDV (模型驱动视图, Model-driven Views)。我们的希望是 Angular 可以作为想法的试验场, 如果发现有价值的想法, 就可以转化为 Web 标准并推动到浏览器中。

3.5 总结

AngularJS 有一种独特的网络应用开发方式。通过使用 JavaScript 代码中的依赖注入和 HTML 模板中的指令, AngularJS 应用是声明式的。双向动态数据绑定意味着所有的数据都自动保持同步, 这样我们就可以把精力放在应用的功能上, 而不是组件间的事件通信上。

3.5.1 AngularJS 的强项

AngularJS 提供了开箱即用的功能集, 有依赖注入容器、解析管理指令的模板编辑器、端到端测试库等特性。这可以让你在服务、控制器、过滤器和其他组织良好的

组件中管理代码。你可以为普通的 HTML 网页添加属性使其动态化，而不需要写很多额外处理代码。最后，你可以写测试来验证它工作是否完好。

AngularJS 提供的功能极大地简化了基本 CRUD 应用的搭建。大量真实网络开发倾向于这种类型的应用，使得 AngularJS 成为很多项目很好的选择。

AngularJS 还有很多潜能。可能是因为它创建于 Google，当然也可能因为它有很多强大的功能，使用 AngularJS 的人十分多，而且还在增长。这也促生了很多在线资源：教程、文章、演讲、博客和论坛问题的回复。

3.5.2 AngularJS 的弱项

虽然这种声明方式很好，但如果你不够细心的话，在大型项目中会很容易失控。如果你滥用了依赖注入，代码可读性会变差。获取低耦合性的代价就是函数调用和函数代码之间的直接访问变少了。相似的，在对指令和表达式足够了解之前，模板会显得很难读懂和维护。

AngularJS 提供了很多很神奇的东西，如依赖注入、双向动态数据绑定、视图指令等。在调试应用程序时，你会觉得四处都是障碍。找出 AngularJS 应用的某些地方不工作的原因可能不是简单的事情，因为它不像其他框架那样底层结构更简单直接。

除此之外，AngularJS 的学习曲线会比其他传统方式的框架更陡一些。如果你遇到困难，找到解决方案会有挑战性。

3.5.3 最后的思考

AngularJS 有很多令人激动的特性。如果它的理念吸引了你，使你决定更加深入地了解它，你就会发现有一个极具热情的大社区供你分享兴趣。

第 4 章

Ring

你可以用一桶乐高积木组装成任何东西，如一所房子、一辆汽车、一艘宇宙飞船，或者只是把所有的积木都堆积起来。虽然每块积木都不相同，但都遵循着同样的拼接方式。这个简单而一致的设计能够使不同的积木很容易地拼接在一起。

Clojure 这样的函数式编程语言就像是一桶乐高积木，每一个组件都很简单，可以很轻松地和其他组件连接在一起完成很多不同的工作。而这些组件的背后也遵循着同样简单的设计，没有使你的程序变得复杂的隐藏机制或组件。

Ring 是一种将 Clojure 中的 Web 函数库连接在一起的方式。就像所有的乐高积木那样，Ring 及其相关的函数库可以通过很多方式构建一个完整的 Web 应用。这样的 Web 应用没有总体设计，只有一些简单的连接规则，所以每个应用都是定制化的。

4.1 Ring 简介

Ring 并不是一个严格意义上的 Web 框架，而是 HTTP 交互的简单的抽象。Clojure 里只有很少的 Web 函数库，甚至没有完整的 Web 框架，而这些仅有的组件就构成了 Ring 的整个基础。在这一章，我们将探索 Ring 以及一些常用函数库。

Ring 将 HTTP 请求和响应抽象为数据，这样的数据很容易被 Clojure 提供的标准

工具处理。在一个为处理数据而设计的语言中，使用 Ring 把操作 HTTP 变成操作数据可使 Clojure 开发者发挥更强大的能力。美国著名的计算机语言科学家 Alan Perlis 曾经说过：“用 100 个函数操作一种数据结构胜于用 10 个函数操作 10 种数据结构。”¹

通过本章，你将看到 Ring 是如何通过处理数据实现 URL 路由、生成 HTTP 响应、中间件以及自动化测试的。不像其他的框架和函数库，Ring 只需要很少的代码就可以工作。它做的仅仅是查找数据字典里的值、在 list 里添加和删除元素等基本的操作，这些基本的操作都在底层进行。

在 Ring 的最上层，我们将会看到 Ring 中处理一般任务的函数库。

我们将使用这些函数库编写一个 BUG 跟踪系统，从而了解这些函数库是如何使用的。最后我们会通过添加 REST 风格 API 和其他常用功能完成它。

在第 1 天里，我们将学习如何用 Hiccup 生成 HTML；Hiccup 会把 HTML 加工成可以被 Clojure 处理的数据。然后我们会用 Korma 访问 SQL 数据库，利用 Korma 能写出可编程且可复用的 SQL。最后用 Compojure 把它们组合起来，告诉 HTTP 请求去哪里访问资源。

在第 2 天里，我们将创建 REST 风格的 API，并通过使用 data.json 函数库处理 JSON 格式数据的输入和输出。为了确保数据的有效性，我们会使用 Valip 验证数据。之后我们继续深入了解 Compojure 是如何处理 URL 路由的。

最后，在第 3 天里，我们将继续深入 Ring 的底层并学习编写自己的中间件。在了解 Enlive——这个世界上最酷的模板函数库的过程中，我们会从不同的角度理解 HTML 是如何作为数据被处理的；同时，我们还会学习使用 clojure.test 库和 Kerodon 来测试 Ring 应用。

虽然涉及了很多内容，但是我们通过一个简单的图就能理解所有组件之间的关系。图 4-1 展示了一个典型的 Ring 应用程序的架构以及每个组件之间的交互关系。

就像 Ring 本身一样，这些组件将自己转换为数据再进行下一步处理。这种独特的编程方式在 Clojure 里很普遍，也同样适用于其他语言。对于构建在 Ring 上的 Web

¹ <http://www.cs.yale.edu/quotes.html>

应用程序而言，所有的任务都是在处理数据。

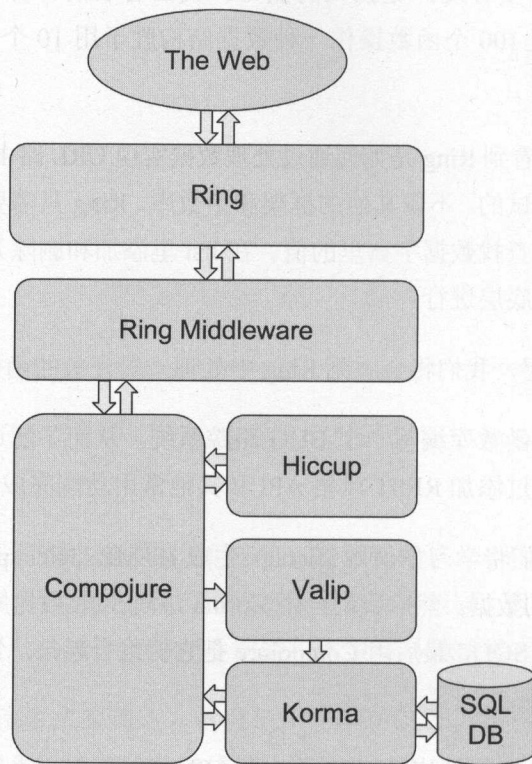


图 4-1 Ring 应用结构

4.2 第1天：基础组件

今天我们要探索 Ring 并构建 Bug 跟踪系统——Zap。就像其他常用的 Bug 跟踪系统一样，Zap 将会记录多个项目的 Bug，允许用户在 Bug 提交记录上添加注释（comment）和状态（status），例如发现（found）和修复（fixed）。Zap 采用非常通用的架构：一个用来存储 Bug 和其他数据的数据库，一组映射视图的 URL 路由，以及一些生成 HTML 的工具。

从今天开始就应该记得，本章探讨的几乎每一个组件都可以被其他组件替换。本周的后几天，你会看到一个组件的替代品。

4.2.1 起步

Clojure 是一个有趣的编程语言，因为它实际上只是一个函数库。你不需要像其他语言那样进行安装和设置，只需要一个 Java 虚拟机（JVM）和 Leiningen 构建工具就够了。

你可以在自己电脑上的包管理工具中找到 JVM，或者从 <http://www.oracle.com/technetwork/java/javase/downloads/index.html> 下载。Leiningen 可以在 leiningen.org 下载。这里我们使用的是 2.x 版本，而不是较早的 1.x 版本。

项目中的所有依赖都会在 Leiningen 的配置文件 `project.clj` 中声明，包括你的应用想使用的 Clojure 版本。

Leiningen 帮你完成所有依赖组件的查找、下载和管理的任务。如果你有使用过 Java 或者其他基于 JVM 编程语言的经验，就会发现 Leiningen 甚至都帮你处理好了 `classpath` 的设置。

Leiningen 这个奇怪的名字是什么？

Leiningen 这个名字源自于 Carl Stephenson 写的名叫“Leiningen VS Ants”的文章。这是一个向 Java 经典构建工具——Ant 发起挑战的一个小故事。Ant 的主要作者——Phil Hagerlber 一直在向 Clojure 社区贡献很多有趣的工具和函数库，这些工具的名字都同样富有想象力。他编写的 `Slamhound`¹ 和 `RobertHooke`² 也非常棒。

Leiningen 会在 Clojars 和 Maven 中心代码库^a 中查找依赖。Clojars 是一个 Ring 应用。如果你想寻找 Ring 在真实项目应用的例子，Clojars 的代码会是个很好的开始^b。

^a <https://clojars.org/> and <http://search.maven.org/>, respectively

^b <https://github.com/ato/clojars-Web>

4.2.2 Hello, World!

你的第一个任务是创建一个新的 Leiningen 项目。无论你在什么目录中存放自己的项目，都只需执行以下的命令就可以创建一个初始项目的模板：

¹ 译注：<https://github.com/technomancy/slamhound>

² 译注：罗伯特·胡克，英国科学家，材料弹性胡克定律发明人，同时也设计了最早的显微镜和望远镜，并著有《显微术》一书 <https://github.com/technomancy/robert-hooke>


```
$ lein new hello
```

Generating a project called hello based on the 'default' template.
To see other templates (app, lein plugin, etc), try `lein help new`.

这个名叫 `hello` 的项目会基于默认的'default'模板生成, 你可以使用'`lein help new`'命令来查看其他的项目模板。

现在需要编辑 `project.clj` 文件来增加你的第一个 Ring 应用的依赖组件, 像下面这样:

```
clojure/hello/project.clj
```

```
(defproject hello "0.1.0-SNAPSHOT"
```

```
  :description "Hello World"
```

```
  ① :dependencies [[org.clojure/clojure "1.5.1"]  
                  [ring/ring-core "1.1.8"]  
                  [compojure "1.1.5"]]
```

```
  ② :plugins [[lein-ring "0.8.3"]]
```

```
  ③ :ring {:handler hello.core/app})
```

① 这些是项目依赖, 其定义格式是 `[[GROUP-ID/ARTIFACTID "VERSION"]]`¹。这些标识符通常定义在组件的 README 文件里。如果 GROUP-ID 和 ARTIFACT-ID 一样, GROUP-ID 就可以被省略。

② `:plugins` 关键字定义了要在这个项目中使用 Leiningen 的插件, 格式和 `:dependencies` 一样。在这个例子中, 我们使用了 `lein-ring`, 它提供了一些用来打包和运行 Ring 应用的工具。

③ `:ring` 的这部分是 `lein-ring` 插件的配置信息。`:handler` 指向在 Ring 应用中定义的最上层的请求处理器。它的值是一个命名空间标识符, 斜线之前是命名空间, 斜线之后是用来绑定应用程序的标识符。

接下来要做的事就是为这个简单的 Web 应用编写代码。

在 Leiningen 项目中, 源代码存放在 `src` 目录下。文件系统的目录结构和命名空间的结构是一致的。例如, `hello.core` 中的代码就会被放在 `src/hello/core.clj` 里。同理, `com.pragprog.7Web.zap.views` 的代码对应的文件就是 `src/com/pragprog/7Web/zap/views.clj`。如果你熟悉 java 的命名空间和包的话, 就会很容易理解它。

¹ 译注: [包/组件 "版本"]

Clojure 的命名问题

在 Clojure 里，符合习惯的命名是用中画线“-”作为分隔符，而不是采用下划线“_”或者首字母大写。例如，你应该使用 `project-name` 而不是 `project_name` 或者 `projectName`。尤其是关键字、标示符和命名空间。

遗憾的是，虽然 Clojure 的命名空间就是 Java 的包（package），但是中划线在 Java 标识符中却是非法的。如果你不仔细命名的话，就会造成一些问题。解决这些问题的方案是在文件名中使用下划线，而在命名空间中使用中划线。比如，命名空间 `hello.people-of-earth` 将会被定义在 `src/hello/people_of_earth.clj` 文件里。

```
src/hello/people_of_earth.clj
```

Clojure 编译器会根据使用场景将这些中画线自动转换为下画线，但是编译器仍然期望文件使用下画线。

Leiningen 的默认项目模板已经创建了 `src/hello/core.clj` 文件，同时定义了 `hello.core` 命名空间。用下面这个最小应用的代码替换文件中样例代码：

```
clojure/hello/src/hello/core.clj
```

```
(ns hello.core
```

```
1   (:use compojure.core))
```

```
2   (defroutes app
```

```
3   (GET "/" []
```

```
    "Hello, World!"))
```

❶ 命名空间中的 `(:use ...)` 会把 `compojure.core` 里所有声明的标识符都放在当前命名空间下，因此它们在当前的命名空间里是可用的。我们即将用到的 `defroutes` 和 `GET` 都在里面。

❷ `defroutes` 定义了应用的 URL 路由信息。和其他在 Clojure 以 `def` 作为开头的定义标识符类似，它用来给标识符绑定路由。注意 `app` 的当前命名空间是 `hello.core`，这意味着在 `projects.clj` 文件里对应的 Ring 插件设置将指向 `hello.core/app`。路由信息会依次列出，路由逻辑将检查每一条路由信息条目，如果不匹配则检查下一条。

❸ 这里列出的唯一一条路由定义了一个指向根路径的 HTTP GET 处理器。方括号表示一个空向量，我们会在之后学习路径参数绑定的时候详细讨论，这里仅返回一个简单的字符串。Compojure 会把 `string` 转换为 HTTP 200 的响应。

剩下要做的就是启动并测试这个应用。现在是 Clojure 发挥 Java 生态系统和 Leiningen 插件能力的时候了。在项目根目录下执行以下命令：

```
$ lein ring server
2013-03-02 21:34:21.040:INFO:oejs.Server:jetty-7.6.1.v20120215
Started server on port 3000
```

这个命令启动了一个监听 3000 端口的本地服务器，同时会打开你的默认浏览器并且跳转到应用的根路径。如果你是在非图形化环境下执行，可以使用 `lein ringserver-headless` 命令跳过打开浏览器的操作。

由于 Leiningen 可能会下载一些应用和启动本地服务器的依赖组件，命令行输出可能和上面的提示信息不同。

如果一切按计划进行，你应该可以在自己的浏览器中看到“Hello, World!”。恭喜你！已经编写了自己的第一个 Ring 应用。接下来，我们将开始创建 BUG 跟踪系统——Zap，同时会学到其他用来构建 Web 应用的函数库。

4.2.3 用 Korma 处理数据

有多少个程序员，就有多少种构建 Web 程序的方式。有的人喜欢从设计 URL 开始，有的人则喜欢从构建用户界面的 HTML 效果图开始，另外一些人则喜欢先组织数据。我们将从数据开始构建出视图和 URL 路由。这样做的好处是：当我们想获得视图的时候，不必创建测试数据，而是直接使用已有的数据模型。

Bug 跟踪系统里应该有什么？

我们假设用户期望 Zap 包含一组特性：同时跟踪多个项目（project）。每个问题（issue）都会有一个标题（title）、一段描述（description）和注释（comments）。一个问题会有不同的状态，例如待处理（open）或已修复（fixed）。这是一个非常简单的设计，但是它应该在第一个迭代后就具备完整的功能。当用户开始把自己的项目放到 Zap 里跟踪后，新需求就会接踵而至。

我们可以简单地用关系模型设计数据模型，我们要用一个关系型数据库存储它。虽然有很多知名的数据库都可以达到我们的目的，但为了减少一些不必要的组件，我们使用 SQLite。

其他语言的框架都能用代码而不是 SQL 创建数据模型。但 Clojure 还没有类似的工具，所以我们不得不编写 SQL 来创建数据库对象。

花点时间思考一下这些不同的部分是如何联系在一起的，然后看看下面这些我们在这一章中即将用到的数据库对象：

```
clojure/zap/day1/resources/data/schema.sql
-- zap schema

CREATE TABLE project (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL
);

CREATE TABLE issue (
  id INTEGER PRIMARY KEY,
  project_id INTEGER REFERENCES project(id) NOT NULL,
  title TEXT NOT NULL,
  description TEXT NOT NULL,
  status INTEGER REFERENCES status(id) NOT NULL
);

CREATE TABLE status (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL
);

CREATE TABLE comment (
  id INTEGER PRIMARY KEY,
  issue_id INTEGER REFERENCES issue(id) NOT NULL,
  content TEXT NOT NULL
);

-- status enums
INSERT INTO status (id, name) VALUES (1, 'open');
INSERT INTO status (id, name) VALUES (2, 'fixed');
INSERT INTO status (id, name) VALUES (3, 'wontfix');
INSERT INTO status (id, name) VALUES (4, 'invalid');
```

除了创建这四张将要用到的表，我们已经在 `status` 表里插入了 4 条 Bug 状态的数据。现在，你可以在项目目录下创建一个数据库加载这些数据对象。目前，大多数操作系统都会默认安装 SQLite。如果你的操作系统没有安装 SQLite，例如 Windows，那就得从 SQLite 的官方网站¹上下载了。

```
$ sqlite3 -init resources/data/schema.sql zap.db .quit
-- Loading resources from resources/data/schema.sql
```

随着新数据库创建完成，我们可以看看如何通过 Clojure 访问它。

¹ <https://www.sqlite.org/>

构建数据模型

我们使用 Korma 访问数据库¹。不像其他框架中的对象关系映射（Object Relational Mapping, ORM）系统，Korma 更贴近 SQL 的概念。但它暴露出来 SQL 是可组合（composable）的，你很快就会看到为什么这个特性非常好用。

首先，我们必须告诉 Korma 数据库的所有信息。Korma 会在数据库实体中访问表，就像 Clojure 里用 `def` 和 `defn` 来定义标识符和函数一样。Compojure 使用 `defroute` 定义路由信息。Korma 则使用 `defdb` 和 `defentity` 定义数据库和数据库实体。

```
clojure/zap/day1/src/zap/models.clj
(ns zap.models
  ❶ (:refer-clojure :exclude [comment])
    (:use korma.db korma.core)
    (:require [clojure.string :as string]))

  ❷ (defdb zap
    (sqlite3 {:db "zap.db"}))

  (defentity project
  ❸ (entity-fields :id :name))

  (declare comment)
  (defentity issue
    (entity-fields :id :project_id :title :description :status)
  ❹ (has-many comment))

  (defentity status
    (entity-fields :id :name))

  (defentity tag
    (entity-fields :id :issue_id :tag))

  (defentity comment
    (entity-fields :id :issue_id :content)
  ❺ (belongs-to issue))
```

❶ 如果你以前没有见过 `:refer-clojure`，会感觉这一行有点奇怪。由于 `ns` 宏会自动把 `clojure.core` 中所有的标识符引入，而 `comment` 标识符会和数据库实体 `comment` 名字冲突，所以这里需要去除对 `comment` 的引用。

❷ 这里定义了数据库，对于 SQLite 来说配置信息十分简单。但如果是在生产环

¹ <http://sqlkorma.com/>

境下的数据库，则需要设置用户名、密码和主机参数。

③ `entity-fields` 告诉 Korma 在查询的时候默认返回哪些字段。当然这不是必需的，不过如果数据库对象未来可能被修改，`entity-fields` 就会非常有用。

④ Korma 支持定义数据库实体间的关系，`has-many` 关系意味着对于每一个 `issue`，都关联着 0 条或者多条 `comment`。这些关联信息可以在查询时自动连接（`join`）对应的表。

⑤ `belongs-to` 是和 `has-many` 相对应的关系。

你看到的这些，可能看起来像是在其他框架中的数据模型的定义。注意这里没有行为，仅有一些数据的描述。

用 REPL 探索 Clojure

在研究数据模型之前，让我们在 REPL（Read Eval Print Loop 读取-求值-打印循环或称交互式编程环境）里和 Korma 做一些交互。你可以在项目目录中执行 `lein repl` 命令启动它，然后使用 `in-ns` 函数切换到 `zap.models` 命名空间下。

```
nREPL server started on port 54315
REPL-y 0.1.9
Clojure 1.4.0
<<help text>>
user=> (require 'zap.models)
<<omitted output>>
nil
user=> (in-ns 'zap.models)
#<Namespace zap.models>
zap.models=>
```

让我们在数据库里增加一些项目信息。像 SQL 一样，Korma 使用 `insert` 命令创建新行。

```
zap.models=> (insert project (values {:name "Zap"}))
<<omitted output>>
{:last_insert_rowid() 1}
zap.models=> (insert project (values {:name "Website"}))
{:last_insert_rowid() 2}
```

`select` 用于执行查询。你可以使用 `fields` 选择返回哪些字段（这样会覆盖 `entity-fields` 中定义默认显示字段）。`where` 的作用和 SQL 中一样，但是在这里 `where` 使用的是 Clojure 表达式。

```

zap.models=> (select project)
[{:name "Zap", :id 1} {:name "Website", :id 2}]
zap.models=> (select project (fields :name))
[{:name "Zap"} {:name "Website"}]
zap.models=> (select project (where {:id 1}))
[{:name "Zap", :id 1}]
zap.models=> (select project (where (or (> :id 1) (= :name "Zap"))))
[{:name "Zap", :id 1} {:name "Website", :id 2}]

```

给 where 传入一个 map 是一个快捷语法。每个键 (key) 是一个字段名, 值 (value) 就是表达式, 不能为空。所有的这些键值对在 where 中必须一一匹配, 你可以参阅 Korma 的文档来了解这方面的详细内容。

where 子句里, 同样可以使用 order、group 和 join, 这些和 SQL 里都是一一对应的。删除行和更新列的操作很像 select, 你将看到我们用这些语句为 Zap 编写的数据库模型代码。

构造数据模型

我们仍然需要给剩下的需求定义数据模型函数。我们需要创建项目 (project)、问题 (issue) 和注释 (comment), 还要列出这些项目和问题, 以及取得每个问题的详细描述并支持修改问题的状态。让我们从和项目相关的函数开始吧。你也可以自己在 REPL 里用这些函数做实验。

```

clojure/zap/day1/src/zap/models.clj
(defn all-projects []
  (select project))

(defn create-project [proj]
  (insert project (values proj)))

(defn project-by-id [id]
  ❶ (first (select project (where {:id id}))))

```

❶ select 语句总是返回一个列表 (list), 即使只返回一行。我们用 first 语句可以得到返回结果, 如果为空则返回 nil。

和问题相关的函数有一点复杂:

```

clojure/zap/day1/src/zap/models.clj
❶ (defn issue-query []
  ❷   (-> (select* issue)
  ❸     (fields [:issue.id :id]
                :project_id
                :title

```

```

      :description
      [:status.id :status_id]
      [:status.name :status_name]))
4 (join status (= :issue.status :status.id)))

5 (defn issues-by-project [id]
  (-> (issue-query)
    (where {:issue.project_id id})
6    exec))

7 (defn issue-by-id [id]
  (-> (issue-query)
    (where {:issue.id id})
    exec
    first))

```

❶ 这是一个返回部分查询结果的助手函数，由于其他模型函数都需要使用右关联（right joined）取得关联数据，所以这些公共的部分可以被提取出来。

❷ 不像 SQL，Korma 的函数允许你以增量的方式构造一个查询，而不用拼接字符串。在这里 Clojure 的线程操作使用 `select*` 构建了一个基本的 `select` 查询，然后给它们传入字段和连接作为参数。其他的 Korma 函数也有类似于 `*-versions` 的东西用于以增量的方式构造查询。虽然其他框架支持类似的方法，但在 Korma 中则显得十分优雅。

❸ 这里传入一个向量而不是关键字，就像给一个字段创建了别名。在这个例子中，问题表的 `id` 字段会在结果列表里以 `:id` 出现。

❹ 连接第二张表同样很简单，这里我们把问题的状态和状态表关联起来。

❺ 我们的查询从返回部分查询结果的助手函数开始。

❻ 部分查询和那些创建的 `select*` 查询一样。但不把这些查询传给 `exec`，它们是不会执行的。

注释（comment）和状态（status）的数据模型函数非常相似。用户要求我们的应用能够查询指定的 Bug。实现问题查找（Find-issues）功能给了我们一个展现 Korma 更多有趣特性的机会。

```
clojure/zap/day1/src/zap/models.clj
```

```

(defn find-issues [q]
  (let [q (str "%" (string/lower-case q) "%")]
    (-> (issue-query)
      (where (or (like (sqlfn lower :issue.title) q)
                  (like (sqlfn lower :issue.description) q)))
      exec)))

```


首先，查询已经构建好。SQL 使用%作为通配符来匹配任意字符串。然后，我们的 `issue-query` 助手函数会被再一次用到。这时发挥了组合的威力。最后注意 `like`，它和 SQL 中的 `like` 一样。`sqlfn` 用于对指定字段调用一个内置的 SQL 函数，这样搜索的时候就可以忽略大小写。现在数据模型可以正常工作了，接下来轮到视图部分了。

4.2.4 用 Hiccup 把数据转化为 HTML

视图的责任是为应用生成 HTML。我们将用 Hiccup 把 Clojure 的数据转变为 HTML¹。Clojure 众多神奇的数据操作函数能把这些单调乏味的工作变得快速高效。

Hiccup 的基本思想很简单。HTML 元素被看作是以元素名作为关键字的向量。元素属性是一个可选的 `map`，子元素或者子元素列表也采用同样的格式。这里有一些例子：

```
clojure/examples/src/examples/hiccup1.clj
[:h1 "Zap Issue Tracker"]

[:div {:class :content}
 [ :p "Do you have issues? Zap can help!"
   :p "Zap is a simple issue tracking solution ..."]]
```

这些看起来有点冗长的语法和你写的 HTML 内容相同，没什么神奇的。关键在于每一个函数返回相似的数据结构，这让你能使用 Clojure 的全部特性来自动化 HTML 的编写。

```
clojure/examples/src/examples/hiccup2.clj
[:ul
 (for [item items]
  [:li (:name item)])]

[:body
 [:div {:class :header}
  (include-header)]

 [:div {:class :content}
  [:h1 "Welcome!"
   :p "..."]]]
```

¹ <https://github.com/weavejester/hiccup>

Hiccup 提供了一个名为 `html` 的宏把数据输出为真正的 HTML。让我们通过 Hiccup 的 `html` 函数运行第一组例子：

```
clojure/examples/src/examples/hiccup1.clj
(require '[hiccup.core :refer [html]])

(html [:h1 "Zap Issue Tracker"])
;;=> "<h1>Zap Issue Tracker</h1>"

(html
 [[:div {:class :content}]
  [:p "Do you have issues? Zap can help!"]
  [:p "Zap is a simple issue tracking solution ..."]])
;;=> "<div class=\"content\"><p>Do you have issues? Zap can help!</p><p>Zap is a
;; simple issue tracking solution ...</p></div>"
```

Hiccup 简单得有点不可思议。将 HTML 转变为数据可以让你最大程度地发挥创造力。你可能想写一个函数用来插入页头页尾，或者重写 URL，或者为每个页面设置不同的过滤器。它们其实就是把元素插入一个向量，对集合类型进行映射，或者过滤并拼接任何序列。

让我们使用 Hiccup 来生成主页：

```
clojure/zap/day1/src/zap/views.clj
1 (defn base-page [title & body]
  (html5
   [:head
    (include-css "/css/bootstrap.min.css")
    (include-css "/css/zap.css")
    [:title title]]
   [:body
    [:div {:class "navbar navbar-inverse"}
     [:div {:class :navbar-inner}
      [:a {:class :brand :href "/" } "Zap!"]
      [:form {:class "navbar-form pull-right"}
       [:input {:type :text :class :search-query :placeholder :Search}]]]]
    [:div.container (seq body)]]]))

2 (defn projects []
  (base-page
   "Projects - Zap"
   [:div.row.admin-bar
    [:a {:href "/projects/new"}
     "Add Project"]]
    [:h1 "Project List"]])
```

```

4      [:ol
      (for [p (models/all-projects)]
        [:li [:a {:href (str "/project/" (:id p) "/issues")} (:name p)]])])

```

❶ `base-page` 是一个用来生成主站的模板的助手函数，它会在对应的地方插入标题和内容。

❷ `Hiccup` 包含了一些助手函数，例如 `include-css` 就可以让你很容易地添加样式。它所做的仅仅是生成了 `<link>` 标记。

❸ `body` 可能是一个列表或者一个向量，`seq` 用于把它们转换为序列。`Hiccup` 把一个向量看作是一个元素，却把序列看作是一个子元素的列表。

❹ 你可以把这里读作“对于每个项目 `p`，生成了一个带有 `p` 的信息的列表项”。

下面的视图用来创建新的问题，注意 `Hiccup` 提供的表单生成函数：

```

clojure/zap/day1/src/zap/views.clj
(defn new-issue [id]
  (let [proj (models/project-by-id id)]
    (base-page
      (str "New Issue for " (:name proj) " - Zap")

      [:h1 "New Issue for " (:name proj)]
      (form-to
        [:post (str "/project/" (:id proj) "/issues")]
        (text-field {:class "span8"
                     :type :text
                     :placeholder "Title"} :title)

        [:br]
        (text-area {:class "span8"
                    :placeholder "Description"
                    :rows 5} :description)

        [:br]
        (submit-button {:class "btn btn-primary"} "Create Issue")))))

```

由于其他的视图生成代码很相似，所以我们在这里就不过多地展示其他类似的视图生成代码了。你可以直接通过 `src/zap/views.clj` 查看剩下的部分。

我们几乎完成了一个完整应用的所有部分，剩下的就是把 URL 映射到对应的视图上了。

4.2.5 使用 Compojure 处理路由

大多数 Web 框架试图抽象出路由逻辑，这样就很容易创建和管理满足不同应用需求的 URL。但在 Clojure 中这样的组件很少，而且其关注点和喜好各不相同。我们将在 Zap 里使用 Compojure，它是最早也是最流行的路由函数库。

除了路由外，我们还必须关注 HTTP 请求是如何被处理的。例如，对于表单参数和查询参数的处理最好采用统一的形式，这样我们就不用分别写代码对它们进行了。基于这个思想，我们来看一下 Ring 的中间件是怎么处理这类问题的。

URL 和视图

在“Hello, World”那一节里，你已经看到了一个 URL 映射到视图的例子。Compojure 中的 `defroutes` 宏会做一系列诸如把 URL 映射转化成代码的事情，这样就可以根据输入的 URL 找出并执行正确的视图代码。最简单的方法是直接实现 Zap 的路由：

```
clojure/zap/day1/src/zap/core.clj
```

```
(defroutes app-routes
```

- ① (GET "/" []
 (view/index))
 (GET "/projects" []
 (view/projects))
 (GET "/projects/new" []
 (view/new-project))
 (POST "/projects" [& params]
 (view/make-project params))
- ② (GET "/project/:id/issues" [id]
 (view/issues-by-project id))
 (GET "/project/:id/issue/new" [id]
 (view/new-issue id))
- ③ (POST "/project/:id/issues" [id & params]
 (view/make-issue id params))

 (GET "/issue/:id" [id]
 (view/issue id))
 (POST "/issue/:id/comments" [id & params]
 (view/make-comment id params))

 (POST "/issue/:id/close" [id & params]
 (view/close-issue id params)))

❶ 这个和你之前看到的路由类似，而在这里返回的是一个函数而不是字符串。除了定义 GET 方法的路由列表外，你还可以看到其他常见 HTTP 方法的路由，例如 POST、PUT、DELETE 等。

❷ 作为函数的声明，路由都带有一个参数列表。这些参数作为关键字会匹配 URL 中的占位符。在这个例子中，关键词:id 作为路径的第二个元素被映射为一个同名的 id 变量，使这个变量在路由体代码中可用。

❸ 有时路由逻辑或者视图需要访问 HTTP 请求中的其他部分数据。查询参数会被作为关键字参数传给函数。你可以用 ¶ms 语法把它们集合到一个 map 里，也可以使用 :as req 把全部的请求数据映射到 req 变量里，这样就可以从请求 map 访问所有的请求数据了。

这些路由定义是 Zap 的全部路由定义。它们并不复杂，大多数是把 HTTP 响应委托给对应的视图函数，同时给函数传入必要的参数。Compojure 支持在路由中做一些数据验证，例如用正则表达式限制参数的格式，或者限制整数的取值范围。你可以参考 API 文档以获得更多的信息。

明天我们会看到更多 Compojure 的高级用法。现在让我们开始处理客户端的数据，例如创建问题时需要填写的信息。

中间件和输入处理

Compojure 允许我们访问路由函数里以 map 形式封装的请求数据。但是这些 map 里有哪些数据？我们如何访问取得它？实际上 map 里包含所有的 HTTP 请求数据，可这些数据不是很容易访问。下面的这段代码就是一个访问含有请求数据的 map：

```
clojure/examples/src/examples/request.clj
```

```
{:remote-addr "127.0.0.1"
 :scheme :http
 :request-method :get
 ❶ :query-string "a=1&b=2"
 :route-params {}
 :content-type nil
 :uri "/foo"
 :server-name "0.0.0.0"
 :params {}}
```

```

:headers {"accept-charset" "ISO-8859-1,utf-8;q=0.7,*;q=0.3"
          "accept-language" "en-US,en;q=0.8"
          "accept-encoding" "gzip,deflate,sdch"
          "user-agent" "Mozilla/5.0 ..."
          "accept" "text/html,..."
          "connection" "keep-alive"
          "host" "0.0.0.0:3000"}
:content-length nil
:server-port 3000
:character-encoding nil
② :body #<Input org.mortbay.jetty.HttpParser$Input@4251b296>}

```

① 查询字符串包含了所有的查询参数，但是完全没解码。

② 输入的数据流可能包含表单参数，但是也没解码。

自己编写从 map 里解码所有数据的程序并不是很有趣。幸运的是，Ring 可以帮你做这类工作。想象自己正在写一个函数，用来把这些 map 变成另外一个容易访问的 map。这个新的 map 里可能会解码出一个 :query-string。我们同样可以用这种方式处理表单数据。你甚至可以给 map 里添加新的键值，让浏览器更容易处理。

Ring 调用中间件函数处理这些请求数据，中间件函数也可以用同样的方式处理响应数据。这些简单的函数组合在一起可以发挥强大的功能，使你能够完成不同的事情。

Ring 已经包含了这个用来处理查询参数和表单参数的中间件。ring.middleware.params/wrap-params 和 ring.middleware.keyword-params/wrap-keyword-params 可以把原始数据转化为 map 参数 :keyword-params 和 :form-params 函数，而且把两者合并为一个更容易使用的 map—— :params，最后把这些所有的键转化为 Clojure 的关键字。

由于 Ring 的中间件由这些简单的数据转化函数组成，我们可以根据 Compojure 的路由信息把这些函数全部串联起来变成一个完整的应用。这就是 Zap 的主应用定义：

```
clojure/zap/day1/src/zap/core.clj
```

```

(def app
①  (-> app-routes
②    (wrap-resource "public")
    wrap-keyword-params
    wrap-params))

```

① app-routes 是 Zap 的路由信息边界，是在 defroutes 中定义的。

② wrap-resource 用来为资源目录里的静态文件服务，Zap 会使用它作为 CSS 和图片等静态资源的存放目录。

我们将在第三天深入探索一些中间件的具体内容，中间件组合的代码读取顺序是自下而上的：首先 `wrap-params` 会解码参数，然后 `wrap-keyword-params` 会把字符串转化为关键字，最后处理和 URL 匹配的静态文件。如果什么都没有找到，应用会再次尝试在应用的路由中查找。当我们的 Zap 路由加载时，含有请求数据的 `map` 就有了很漂亮的结构。

我们现在有了更容易访问的查询表单和表单参数。下面让我们编写创建新的问题的视图。

```
clojure/zap/day1/src/zap/views.clj
(defn make-issue [id params]
  (let [iss (merge params {:project_id id :status 1})]
    (models/create-issue iss)
    (response/redirect-after-post (str "/project/" id "/issues"))))
```

这段代码为用户的输入数据添加了一些默认的参数，然后把它传递给对应的数据模型函数来创建问题。由于这是个 HTTP POST 方法，所以我们要生成一个返回问题列表重定向指令，而不是一个普通的 HTTP 响应。Ring 包含了一些 HTTP 响应生成器，它们都放在 `ring.util.response` 命名空间下。

4.2.6 我们在第 1 天学到的

万事开头难，但在第 1 天你就已经有始有终了，了不起！如果你之前还没有太多的 Clojure 编程经验，这些内容对你来说还是有一些挑战的。你可以通过使用 REPL 学到更多的东西以弥补这方面的不足。Clojure 程序员经常构建一些显而易见的简单函数，把整个工程构建得更加优雅。今天看到的大部分组件，核心概念都是简单的数据转换。这一点会让你通过交互的方式探索起来更加简单。

我们很快就通过非常函数化的方式了解了 Ring 应用的基本结构。从数据设计开始，我们采用 Korma 构建数据模型——一个将 SQL 转换为可组合形式的函数库。Hiccup 让创建视图简单到只需要写下它们的数据结构。Compojure 把 URL 请求匹配到对应的视图，并且把所有部分连接在一起。最后我们探索了 Ring 的中间件，它帮助我们z把用户数据转化为方便使用的结构。

每个组件的主要操作就是处理数据。Clojure 的工具箱相较于其他语言可谓非常简陋，但是把每个工具组合在一起使用的威力却是无可匹敌的。

第 1 天的自学

查阅

- James Reeves 在 Clojure Web 开发的博客和他的演讲。
- 其他在 GitHub 上的 Ring 应用。
- 其他可替代的路由库。

实践

- 添加在项目页面上 BUG 数量统计。
- 添加编辑项目，bug 和注释的功能。
- 尝试一下 Korma 的可组合查询。
- 把数据库换成 MySQL 或者 PostgreSQL。

4.3 第 2 天：拼接的模式

玩乐高积木的高手们都有一套自己的架构模式，能把看起来简单的积木组合成特别的设施。

一些特定的积木允许你构建小路，妥善放置的铰链可以创建轴的接合面，普通积木的特殊用法可以构建一个更有视觉冲击力的模型。Clojure 就是一个将简单的组件合成为强大模式的语言，这些模式可以帮助你构建强大的 Web 应用。

下面让我们利用 Clojure 的常见模式构建 Zap 的公共 API。

4.3.1 定义 API

在构建 API 之前，要对接下来要做的事情有一些计划。假设你已经有一些使用或者创建 REST 风格 API 的经验，所以我们直奔主题——实现。

我们需要采用某种方法列出项目以及问题，也会用同一种方法取得项目和问题的

详细信息；同时，我们也要创建、修改、删除这些内容。

让我们从对项目的管理开始，如表 4-1 所示。然后，我们可以定义一些对问题的管理，如表 4-2 所示。

表 4-1 项目操作 API

请 求	描 述
POST /projects	添加一个项目 Add a new project
GET /projects	列举所有的项目 Enumerate all projects
GET /project/ID	根据项目 ID 取得项目的详细信息 Get details for project ID
DELETE /project/ID	根据项目 ID 删除项目 ID Delete project ID
PUT /project/ID	根据项目 ID 更新项目详细信息 Update project details for ID

表 4-2 问题管理 API

请 求	描 述
POST /project/PID/issues	对一个项目创建一个问题 Create an issue for a project
GET /project/PID/issues	列举一个项目的所有问题 Enumerate all issues for a project
GET /project/PID/issue/IID	根据问题的 ID 取得问题的详细信息 Get details about issue IID
DELETE /project/PID/issue/IID	根据问题的 ID 删除问题 Delete issue IID
PUT /project/PID/issue/IID	根据问题的 ID 更新问题详细信息 Update details for issue IID
POST /project/PID/issue/IID/comments	根据问题的 ID 添加注释 Add a comment for issue IID
DELETE /project/PID/issue/IID/comment/CID	根据问题的 ID 和注释删除注释 Delete comment CID

注释将会和问题的详细信息一同返回，而不是单独返回。所以，没有必要为它写一个单独的 GET 方法。在所有的情况下，API 都会返回 JSON 格式的数据并允许接受标准的表单参数。

确定了这些调用 API 的规范，就可以开始实现它们了。我们要从学习如何让 Ring 应用处理 JSON 数据开始。

4.3.2 处理 JSON

和最近其他流行的编程语言一样，Clojure 的生态系统包含很多用来处理 JSON 格式

数据的函数库。多亏它对 Java 的操作支持，我们才可以直接访问 Java 处理 JSON 的类库。

当我们构建基于 JSON 的 API 的时候，会需要用到一些方法把 Clojure 的数据转换为 JSON。最直接的方法是使用 `data.json` 库，它是 Clojure 的 `contrib` 库的一部分。`data.json` 非常灵活，而起关键作用的函数却是 `read-str` 和 `write-str`。`read-str` 读取一个 JSON 字符串并返回对应的 Clojure 数据结构，而 `write-str` 则把 Clojure 数据输出为 JSON 字符串。

让我们看一个例子：

```
clojure/examples/json.clj
(require '[clojure.data.json :as json])
(json/write-str {:foo 1 :bar 2})
;;=> "{\"foo\":1,\"bar\":2}"

(json/read-str "{\"foo\":1,\"bar\":2}")
;;=> {:foo 1 :bar 2}
```

我们昨天创建的数据模型函数和今天要创建的 API 很相像，都可以直接返回 Clojure 数据结构。这意味着为 Zap 创建一个只读的 API 只需要调用之前创建好的模型函数，然后把输出转化为 JSON。

让我们从定义项目列表的 API 路由开始。在 `zap/core.clj` 文件中增加下面的 `defroutes` 代码：

```
clojure/zap/day2/src/zap/core.clj
(GET "/api/projects" []
  (json/write-str (models/all-projects)))
```

你只需要在昨天创建的 `defroutes` 里增加新的路由项。但这一次请求处理函数会调用合适的模型函数并把它们转化为 JSON 作为返回结果，而不是返回 HTML。

另外一个只读的 API 调用非常相似：

```
clojure/zap/day2/src/zap/core.clj
(GET "/api/project/:id" [id]
  (if-let [proj (models/project-by-id id)]
    (json/write-str proj)
    {:status 404 :body ""}))
(GET "/api/project/:pid/issues" [pid]
  (json/write-str (models/issues-by-project pid)))
(GET "/api/project/:pid/issue/:iid" [pid iid]
  (json/write-str (models/issue-by-id iid)))
```

注意这些路由体中重复的部分，可以在明天学习编写我们自己的 Ring 中间件之后重构它们。现在这些只读的部分已经完成，下面开始添加、修改或者删除数据。

4.3.3 验证输入

为了接受 API 消费者的输入，我们需要验证这些数据以确保它们存储到数据库之前是正确的。一个方法是在 API 视图中编写一系列的验证语句用来测试数据中出现的各种错误。例如项目的名称是否为空，Bug 的状态是否为合法值，问题的所有必填字段是否有值？

实现这个目标会碰到一些问题。第一个问题就是验证数据的代码可能在不同的地方重复，例如创建项目和编辑项目的动作都需要验证与输入的项目数据。第二个问题是这样的方法会形成一个很大的表达式组合树，给后期的阅读和维护带来麻烦。

解决上述问题的一般方案是提取出验证路径，把它们放到函数中去，例如 `valid-project?` 和 `valid-issue?`。但是，这只解决了第一个问题。

Clojure 的解决方案是，使用 Clojure 的能力把组合在一起的 if 表达式抽象出来。想象一下像下面这个指令语言一样，你只需要定义一个规则列表，每一条规则都包含一个表单域名、一个简单的验证规则和一个错误信息。当发现对应表单域的输入数据违反了某条规则时，就返回对应的错误信息。

```
(def validator valid-project?
  [:name present? "name must be specified"]
  [:name (min-length 1) "name must not be blank"])

(valid-project? {:foo "bar"})
;;=> {:name ["name must be specified" "name must not be blank"]}
```

这比仅仅提取出验证逻辑要好很多。现在它返回了一个易于使用的错误列表，你可以利用它为 API 消费者或者用户创建一个更友好的错误提示。

这种分解问题的方式在 Clojure 这样的函数式编程语言里非常常见，把不同的部分组合起来可以得到解决问题的新颖方法。

设计这样一个启发式的 DSL 在实践中是非常简单的。用一个简单的 Clojure 宏拓展了语法，验证逻辑也很方便的在检查每个规则的同时构建出了错误提示信息。幸运

的是，这样的 DSL 已经存在于 Clojure 中了。我们在 Zap 中使用 Valip，另一个由 James Reeves 编写的库¹。

使用 Valip

Valip 由两个简单的部分组成。第一部分是验证函数，这个函数需要一个 map 结构的验证数据和验证的规则作为参数；第二部分则是应用这些规则集中声明的断言。

让我们把前面的例子转换成 Valip 方式：

```
clojure/examples/src/examples/valip.clj
(require '[valip.core :refer [validate]]
         '[valip.predicates :refer [present? min-length]])

(defn valid-project [proj]
  (validate proj
    [:name present? "name must be specified"]
    [:name (min-length 1) "name must not be blank"])))
```

除了 `present?` 和 `min-length` 外，Valip 还有一些其他的断言，例如最大长度 (`max-length`)、正则表达式匹配 `url?` `numeric?`、`between`，甚至是 `dns-lookup`（用于验证给定的主机名是否可以被解析）。当然，编写自己的断言同样十分简单。下面是 Valip 中实现最小长度验证的源代码：

```
clojure/examples/src/examples/valip.clj
(defn min-length
  "Creates a predicate that returns true if a string's length is greater than
  or equal to the supplied minimum."
  [min]
  (fn [s] (>= (count s) min)))
```

（注释：创建一个断言，如果一个字符串长度大于或者等于给定的最小长度值就返回 `true`。）

让我们来创建一些 Zap 的验证规则：

```
clojure/zap/day2/src/zap/validations.clj
(ns zap.validations
  (require [valip.core :refer [validate]]
           [valip.validations :refer [present?]]))

(defn valid-project? [proj]
```

¹ <https://github.com/weavejester/valip>


```
(validate proj
  [:name present? "name must be specified"]
  [:name (min-length 1) "name must not be blank"]])

(defn valid-issue? [iss]
  (validate iss
    [:title present? "title must be specified"]
    [:title (min-length 1) "title must not be blank"]
    [:description present? "description must be specified"]
    [:description (min-length 1) "description must not be blank"]
    [:status (between 1 4) "status id must be between 1 and 4"]]))
```

验证注释将作为你今天的作业。

现在可以对输入的数据进行验证了，剩下的就是在我们的 API 视图里使用它们。

完成 API

我们的任务再一次变得简单，这多亏昨天编写的模型函数代码。让我们从创建项目视图（create-project view）开始，通过向/api/project 发送一个 POST 请求创建一个新的项目：

```
clojure/zap/day2/src/zap/views.clj
(defn create-project [params]
  (let [errors (valids/valid-project? params)]
    (if errors
      {:status 400
       :body (json/write-str {:errors errors})}
      (do
        (models/create-project params)
        {:status 200 :body ""}))))
```

首先，执行校验函数并收集错误信息。如果输入数据验证失败，就会返回一个 HTTP 状态为 400 的错误请求（Bad Request）响应，同时返回一个 JSON 格式的错误信息列表。如果一切正常，项目就会被创建。

让我们来看看有点复杂的部分：编辑项目（edit-project view）视图。和创建项目不同，这个视图可能会引入一个错误的项目 ID。对于这种情况，API 需要返回一个 404 找不到当前页（Not Found）的错误消息。而对于其他的部分来说，处理逻辑几乎是一样的：

```
clojure/zap/day2/src/zap/views.clj
(defn edit-project [id params]
  (let [errors (valids/valid-project? params)]
```

```
(if errors
  {:status 400
   :body (json/write-str {:errors errors})}
  (if-let [proj (models/project-by-id id)]
    (do
      (models/update-project id params)
      {:status 200 :body ""})
    {:status 404 :body ""})))
```

这里应当注意，如果我们的 Bug 跟踪系统本身就很复杂，`valid-project?` 就会对数据做更多复杂的验证。

而这里的代码无须修改太多就可以支持更复杂的验证（可能仅需提供一个更友好的错误消息），我们的验证逻辑列表依然会很清晰，尽管看起来有些长。

剩下的 API 视图和这些项目相关的视图相似。想想你会怎么把这些内容抽象出来，Clojure 会让它变得更容易。

API 视图已经完成了，现在要回顾一下路由看看我们是不是能够做一些改进。

4.3.4 可组合的路由

昨天我们已经使用 `defroutes` 宏为 Zap 创建了一组 URL 路由规则。为 Zap API 添加新的路由的第一种方式就是向路由列表中添加新的路由规则，这是我们目前采用的方式。而对于大型的应用，这个列表会变得非常长。如果有一种方式可以把路由逻辑通过不同的代码片段组合在一起，就能完成复杂的路由了。`defroutes` 宏就有一个支持这种功能的语法：上下文路由（the context route）。

上下文的参数与 GET 和 POST 请求所需要的参数一样。不同的是在这里不直接编写路由动作，只需要传入你创建在别处的路由名就可以了。`defroutes` 宏创建的 Ring 处理器可以直接在路由体中使用。

上下文路由背后的实现方式非常简单。它仅仅测试路径前缀是否匹配第二个参数，就会把前缀从路径中去掉，然后把修改过的 HTTP 请求作为最后一个参数传递给处理器。

让我们看一个简单的例子：

```

(defroutes sub-routes
  (GET "/bar" [] "Bar")
  (GET "/baz" [] "Baz"))

(defroutes app-routes
  (GET "/" [] "Root")
  (context "/foo" [] sub-routes))

```

首先，这里已经有一组定义好的路由规则：`sub-routes`，用来处理对应的 GET 请求。上下文语法被用来把这组规则包含到 `/foo` 路径下。现在请求 `"/"` 路径会返回 `Root`，请求 `/foo/bar` 会返回 `Bar`，请求 `/foo/baz` 则会返回 `Baz`。

注意，在 `sub-routes` 里的路由不需要知道自己处理的路径在哪里结束，它自己会找到的。如果以后你想把这组路由移至 `/legacy/foo` 下，只需要修改它的前缀就可以了。一个复杂的应用可能会定义几组不同的路由规则，每一组规则都相对比较简单。而主应用路由就是把这些上下文的定义列表组合在一起。

虽然 Zap 看起来相对简单，我们仍把路由逻辑分成两组。第一组就是我们昨天创建的路由规则，第二组就是 API 路由。最后 `app-routes` 将把两组路由逻辑，静态资源文件和错误处理器组合起来。

```
clojure/zap/day2/src/zap/core.clj
```

```

(defroutes api-routes
  (GET "/projects" []
    (json/write-str (models/all-projects)))
  (GET "/project/:id" [id]
    (if-let [proj (models/project-by-id id)]
      (json/write-str proj)
      {:status 404 :body ""}))
  (GET "/project/:pid/issues" [pid]
    (json/write-str (models/issues-by-project pid)))
  (GET "/project/:pid/issue/:iid" [pid iid]
    (json/write-str (models/issue-by-id iid)))

  (POST "/projects" [& params]
    (views/create-project params))
  (DELETE "/project/:id" [id]
    (views/delete-project id))
  (PUT "/project/:id" [id & params]
    (views/edit-project id params)))

(defroutes app-routes
  (GET "/" []
    (views/index))
  (GET "/projects" []

```

```

    (views/projects))
  (GET "/projects/new" []
    (views/new-project))
  (POST "/projects" [& params]
    (views/make-project params))

  (GET "/project/:id/issues" [id]
    (views/issues-by-project id))
  (GET "/project/:id/issue/new" [id]
    (views/new-issue id))
  (POST "/project/:id/issues" [id & params]
    (views/make-issue id params)))

(defroutes all-routes
  (context "" [] app-routes)
  (context "/api" [] api-routes))

(def app
  (-> app-routes
    (wrap-resource "public")
    wrap-keyword-params
    wrap-params))

```

现在两个部分已经完全分开了，每一节都只包含和自身有关的所有条目。而与自身无关的条目，例如静态资源和请求处理全部已经被这些组合在一起的部分处理了。

注意，它并不是通过一个函数调用一堆助手函数。路由是可以被组合的，它们可以相互调用，或者委托给其他函数，或者被其他路由组合，或者被 Ring 中间件包装。由于每个路由自己都是一个 Ring 处理器，你可以直接注入，给它们传入一个请求 map 然后返回一个响应 map。这个特性让这些函数很容易测试。

这种组合的方式并不仅仅可以在路由中使用。Korma——我们昨天使用的 SQL 函数库，也把 SQL 变成可组合的形式，允许重用 WHERE 子句和字段定义。这里无须做过多字符串的操作，可以仅仅把两个简单的部分组合成一个更大的组件，让它更易于重用或委托。

组合和它的补充，关注点分离，是 Clojure 应用程序的一个核心概念。Rich Hickey 在他的名为“简单让你轻松”¹的演讲里使用“复杂”这个词，用来描述代码交错或交织在一起而显得不简洁。通过分离复杂的组件，你简化了你的代码，使独立的片段变得可重用并可组合成新的功能。想想我们是如何使用简单的乐高积木组合成一个辆汽车

¹ <http://www.infoq.com/presentations/Simple-Made-Easy>

的。你可以很简单地通过积木创造一辆汽车，但是用汽车去构造其他的组件就很困难了。

4.3.5 我们在第2天学到的

Ring 的整个框架栈就是一组简单、强大、可复用的组件，通过对这些组件的组合来解决不同的问题。今天我们看到了更多的组合模式，例如使用上下文设计 Zap 的公共 API。

我们定义了 Zap 的 API 应该包含什么样的方法和路径，这些方法都遵循通用的最佳实践。

我们看到在 Clojure 里处理 JSON 数据是多么的简单。由于 Clojure 包含一系列丰富的处理数据结构和数据结构文本的工具，所以在 Clojure 里处理 JSON 变得异常的轻松。它们的强大和方便超出了 JSON 自身。

我们也学到了如何使用 Valip 验证输入数据。Valip 通过 DSL 解耦验证逻辑让验证输入数据变得非常简单。

最后，我们展示了如何用小且独立的路由组合成一个完整的路由从而方便地实现更复杂的路由。

明天我们将看另外一个生成 HTML 的方法，并会更深入地了解 Ring 的处理器和中间件。当然，会有更多组合的例子在等着我们。

第2天的自学

查阅

- data.json API 文档。
- Compojure api 文档。
- 可替换的数据验证库。

实践

- 实现注释验证。

- 完成剩下的 API 视图。
- 重构 API，让它支持 XML 和 JSON 两种输出格式。
- 增加支持分页的 API 偏移和限制参数（提示：可以看下 Clojure 的 take 和 drop 函数）。

4.4 第3天：构建应用的其他方法

我们从一开始就使用了大部分 Ring 应用程序都会用到的典型函数库。现在我们会开始探索一个可替换的模板函数库，帮自己从丰富的替代品中做出选择。Clojure 程序员们已经从其他的语言和框架中提炼了一些思想，同时开创了一条搜索 Web 编程的捷径。

你在之前几天已简单了解到 Ring 中间件，今天我们会探索得更加深入，以便能够编写出自己的中间件。Ring 中间件采用的这种模式在其他的 Clojure 函数库里也很常见，尤其在 nREPL（带有网络函数库的 REPL）里是非常重要的。

虽然 Hiccup 对 Clojure 程序员来说是一套便捷的模板引擎，可是它对设计师来说却并不友好。我们会学习到 Enlive——Clojure 里另外一个著名的模板引擎，看看它是如何解决这个问题的。此外，测试是非常重要的，我们会编写一些基本的测试。让我们开始吧！

4.4.1 Ring 中间件

在 Ring 旅程的第一天，我们就建议过把 Ring 中间件看作是修改请求响应 map 的简单数据转换函数。实际上它有一点复杂，这些 map 不会直接被转化。我们在构造一条数据转化的流水线，当请求被处理时它就会执行。因为 Clojure 是一个函数式编程语言，所以你应该不会对中间件实际上操作函数感到惊讶。

从函数式编程的定义，你就知道函数的重要性。在函数式编程语言里，函数自身就是一等公民，如同整数或对象。这意味着函数可以作为返回值，也可以作为变量值，甚至可以作为其他函数的参数。这是个非常强大的概念，你可能已经从其他的函数式编程语言中体会到了这些从 Lisp 借鉴而来的想法。

让我们看一些函数返回函数的例子。为了让这个例子更直接，我们从一个简单的目标开始：写一个函数，把输入的列表修改为关键词。

```
clojure/examples/src/examples/function_return.clj
```

```
(defn keywordize [l]
  (for [elem l]
    (if (string? elem)
      (keyword elem)
      elem)))

(keywordize [1 2 "foo" 3 "bar"])
;;=> (1 2 :foo 3 :bar)
```

非常直接，是吗？想象你已经拥有了一些转化为列表的函数，也可以把它们“关键词化（keywordize）”。简单！只需要把原先的函数返回值作为结果继续传给“关键词化”函数。

```
clojure/examples/src/examples/function_return.clj
```

```
(keywordize (other-transform [1 2 "foo" 3 "bar"]))
;; or
(-> [1 2 "foo" 3 "bar"]
    other-transform
    keywordize)
```

如果你没有准备好输入数据，这里会出问题。所以我们给它传入一个转化函数而不是数据，这样就可以通过添加转化函数得到一个新的转化函数，同时这个函数也可以被其他部分使用。当输入数据可用的时候，我们就可以调用它了。为此，我们需要返回一个函数而不是简单地执行一个函数。

```
clojure/examples/src/examples/function_return.clj
```

```
(defn wrap-keywordize [f]
  (fn [l]
    (keywordize (f l))))
```

如果 wrap-keywordize 被其他转化函数调用，它会返回一个新的函数，这个函数会先执行原来的转化操作，然后在结果上继续执行 keywordize。你将会注意到 keywordize 在 f 的输出上进行操作，但是怎么对它的输入进行操作呢？这时 f 的转化期望字符串被键替代。

```
clojure/examples/src/examples/function_return.clj
```

```
(defn wrap-keywordize-output [f]
  (fn [l]
```

```
(keywordize (f l)))

(defn wrap-keywordize-input [f]
  (fn [l]
    (f (keywordize l))))
```

以上是-output 和-input 版本的包装器，我们现在可以以任意方式包装其他的函数。现在已经没有什么能够阻止你同时对输入和输出的内容进行包装了。这个函数的操作理解起来可能会有点困难，所以你可以通过每一步的改进去理解它。从执行函数到返回函数的改动虽然看起来很简单，却非常强大。它不像给飞机加上翅膀，而是给任何东西加上飞行的能力。

如果你已经理解它们了，恭喜你，因为你理解了 Ring 中间件是如何工作的。当然，Ring 会传入带有请求数据的 map 而不是 list 作为函数的输入，同样替代 list 作为输出的是转化输出并生成一个带有响应数据的 map。Ring 中间件可以选择转化一种或者同时转化两种 map，就像我们的包装函数处理输入和输出一样。

我们可以用中间件处理 Zap 中的一个问题：所有 API 路由都需要通过序列化把响应数据转化为 JSON，利用中间件包装 API 函数就可以让中间件来处理转化的问题。先简单思考一下你可能会怎么做，然后看看下面的方案：

```
clojure/examples/src/examples/function_return.clj
(defn wrap-json-response [f]
  (fn [req]
    (let [resp (f req)
          body (:body resp)]
      (assoc req :body (json/write-str body)))))
```

给 Ring 中间件添加 wrap-json-response 会把普通的 Clojure 响应数据结构转化为 JSON，我们就不再需要为每个路由调用 json/write-str 了。

4.4.2 Enlive

让我们从 Ring 的核心爬到 Ring 的顶端——HTML 模板引擎。在 Hiccup 中，“HTML 就是数据”的思想让我们通过 Hiccup 把 Clojure 的数据结构写成 HTML。下面让我们看下 Enlive 是怎么把已存在的 HTML 内容改造成模板的。

Enlive 是一个 HTML 的折纸艺术，它非常聪明和漂亮。你首先创建一个 HTML 的

效果图，然后就可以把这个效果图直接作为 HTML 模板，通过简单的修改从而得到你想要的输出。例如，效果图的 HTML 包含一些虚构的项目条目。在转化中，第一个会被作为一个模板放入单个项目的数据，然后为每一个项目创建一个列表项，替代效果图中虚构的项目条目。

更好的是，这些转化是用 CSS 选择器和 Hiccup 类似的数据结构写成的。用一些 HTML 模板片段部分替代整个 HTML 文档不同的是，模板可以从效果图的输出中提取出来并可以在任何地方重用。HTML 设计师除了 HTML 外不需要使用其他的任何东西，而且他们的效果图再也不需要被转换成模板。你只需要写下转化函数把需要的改变模板转化为真实的输出。

在转化任何东西之前，都需要一些 HTML 让我们在上面工作。假设我们有一个超棒的设计师朋友已经交给了我们一个 Zap 的项目列表界面：

```
clojure/zap/day3/resources/templates/projects.html
```

```
<!DOCTYPE html>
<html>
  <head>
    <link href="/css/bootstrap.min.css" rel="stylesheet" type="text/css">
    <link href="/css/zap.css" rel="stylesheet" type="text/css">
    <title>Projects - Zap</title>
  </head>
  <body>
    <div class="navbar navbar-inverse">
      <div class="navbar-inner">
        <a class="brand" href="/">Zap!</a>
        <form class="navbar-form pull-right">
          <input class="search-query" placeholder="Search" type="text">
        </form>
      </div>
    </div>

    <div class="container">
      <div class="row admin-bar">
        <a href="/projects/new">Add Project</a>
      </div>

      <h1>Project List</h1>

      <ol>
        <li>
          <a href="/project/1/issues">Zap</a>
        </li>
        <li>
```

```

    <a href="/project/2/issues">Website</a>
  </li>
</ol>
</div>
</body>
</html>

```

Enlive 提供了一个叫作 `deftemplate` 的宏，它可以从一个 HTML 文件和对应的转化规则列表创建一个模板函数。下面让我们通过一个简单的模板看看页面标题是怎么改变的。

```

clojure/examples/src/examples/enlive.clj
(use 'net.cgrand.enlive-html)

(deftemplate page-with-title "templates/projects.html"
  [title]

  [:title] (content (str title " - Zap"))))

```

`deftemplate` 和 `defn` 有点相似，它需要一个新函数的名字，模板的路径（相对于项目资源目录），传递给函数的参数，最后是函数体。模板体由一组转化函数组成，每个转化函数都包含一个 CSS 选择器和转化函数，CSS 选择器作为向量的关键字。这个例子从所有元素中选择了标签为 `title` 的元素，然后转化函数替换这个元素的内容。

Enlive 支持大多数 CSS 选择器，选择器 `#content.big` 在 Clojure 里要被写成 `[:#content :.big :a]`。Enlive 的选择器语法有很多内容¹，会让你更容易定位到自己想转化的元素。

除了函数的内容，Enlive 也提供了 `set-attr` 用来设置元素的属性，`add-class` 用来给元素添加新的 class。`removeattr` 和 `do->` 会把不同的转化函数立刻串联起来。你同样可以把这些函数中的任何一个通过像 `if` 或者 `cond` 这样普通的表达式包装起来，以获得动态的行为。

你也可以在 Enlive 里创建片段（`snippets`），片段和模板类似，但片段是从其他的文档中提取的。让我们创建一个片段，以便可以重用项目列表里的 `` 元素。

¹ <http://enlive.cgrand.net/syntax.html>

```
clojure/examples/src/examples/enlive.clj
```

```
(defsnippet project-item
  ❶ "templates/projects.html" [:.container :ol [[:li first-child]]
    [proj]

  ❷ [:a] (do->
    (set-attr :href (str "/project/" (:id proj) "/issues"))
    (content (:name proj))))
```

❶ `deftemplate` 和 `defsnippet` 的唯一区别就是需要传入一个额外的选择器，从 HTML 提取需要的元素。注意在片段的定义里的新选择器标记 `[[:li first-child]]`，它等价于 CSS 里的 `li:first-child`。为了在 Enlive 里表达这个组合的选择器，需要用自己的向量包装自己。Enlive 的 `first-child` 选择器只是其中一个选择，其他的断言例如 `nth-child attr?`（选择第 *n* 个子节点的属性）和 `text-pred`（选择部分文本）也很有用。

❷ 这里在 `action` 的 `do->` 把多个转化函数结合成一个。

它覆盖了 Enlive 的基础知识，让我们利用它重写 Zap 的整个 `project` 列表视图。

```
clojure/zap/day3/src/zap/views.clj
```

```
❶ (deftemplate base-page "templates/projects.html"
  [title & body]
  [:title] (content title)
  [:.container] (content body))

(defsnippet admin-bar
  "templates/projects.html" [:.container :.admin-bar]
  [links]

  ❷ [:a (but first-child)] nil
  ❸ [:a] (clone-for [[url title] links]
    (do->
      (set-attr :href url)
      (content title))))

(defsnippet project-item
  "templates/projects.html" [:.container :ol [[:li first-child]]
  [proj]
  [:a] (do->
    (set-attr :href (str "/project/" (:id proj) "/issues"))
    (content (:name proj))))

(defn projects []
  (base-page
    "Projects - Zap"
```

```

4 (admin-bar {"/projects/new" "Add Project"})
  (html [:h1 "Project List"])
  (map project-item (models/all-projects)))

```

❶ 基本页面从效果图中创建，替换一些关键的部分。

❷ 这里的 `but` 指令是一个否定指令。除了第一个子节点以外，其他的节点都被删除了。这样就只有一个节点，可以通过下一条规则复制出同样的节点。

❸ `clone-for` 通过第一个元素复制出了集合中的每一个元素。它是 Clojure 中 `for` 语法的扩展，复制了除了 `body` 元素以外的所有元素。

❹ `html` 在 Enlive 里使用 Hiccup 语法创建了新的 DOM 节点。

最终结果看起来比 Hiccup 的直接代码啰嗦一点，但是设计师可以完全在熟悉的环境下独立地工作。集成工作只需要遵循页面的结构。如果页面结构修改了，只需要修改相关的 CSS 选择器就可以得到新的页面。

4.4.3 关于测试

在 Ring 旅程的最后一部分，让我们简单了解一下测试 Ring 应用的方法。Clojure 有很多非常不错的测试组件，包括用来做单元测试的 `clojure.test` 和生成随机测试用例的 `test.generative`。对于测试 Ring 应用来说，我们使用 Kerodon¹。

Kerodon 使我们在 Ring 应用中进行交互测试更加方便。你可以模拟浏览页面，填写表单，单击按钮甚至是重定向。这里有一个小例子：

```

clojure/zap/day3/test/zap/basic.clj
(deftest projects-page-exists
  (-> (session zap/app)
    (visit "/projects")
    (has (status? 200) "page exists")
    (within [:h1]
      (has (text? "Project List") "header is there"))))

```

你可以用 `lein test` 运行自己的测试。

¹ <https://github.com/xeqi/kerodon>


```
$ lein test
lein test zap.basic

Testing zap.basic

Ran 1 tests containing 2 assertions.
0 failures, 0 errors.
```

这个测试访问了/projects，检查 HTTP 响应的状态代码，然后确保已知的文本在页面上得以正确显示。

在这些场景的背后，Kerodon 根据你的行为构建了 Ring 请求，然后验证这些请求的响应映射是否满足测试期望条件。还有一些其他的测试函数库可以用来测试 Ring，你可以从这些不同风格的测试函数库中选择所爱。

4.4.4 我们在第 3 天学到的

今天我们快速回顾了一下 Ring 生态圈里一些有意思的东西。Clojure 社区的创新非常迅猛，解决同一领域问题的工具库很多。了解这些工具的内容已经超出了一本书，更别说一个章节，但是你应该对其他的工具有了一些基本的认识。

我们看到了 Ring 中间件，它在整个应用栈里起着十分重要的作用，串联起了所有 HTTP 请求和响应的转化函数。Ring 中间件展示了 Clojure 组合能力的强大之处。如果你无法在这个级别进行抽象，Clojure 里的组合模式就会让你很困扰。

相较于其他框架中的传统模板，Enlive 显得有点超前，但是对解决问题的能力不弱。特别是对于项目组里由非程序员写出的 HTML，它把 CSS 选择器表达为数据，让你很容易就能操作 HTML。

最后，但并非最不重要，我们看到了测试的一种方法。由于 Ring 的应用中都是传入和返回 map 的简单函数，即使没有 Kerodon 这样强大的库，测试它们依然令人惊奇的简单。

第 3 天的自学

查阅

- 更多的 Enlive 教程。
- 比较 Hiccup、Enlive 和其他的模板框架。

实践

- 从 Ring 的中间件里提取出列表。
- 用 Enlive 转化更多的视图。
- 为增加和关闭项目问题编写测试。

4.4.5 对 James Reeves 的采访

James Reeves 是一家只有一个人的软件公司，同时也是 Clojure Web 生态系统的驱动力。如果你使用 Clojure 编写 Web 应用而没有用到他编写的库，那一定会感觉很吃力。他创建和维护了我们在这一章中用到的一些库，包括 Ring、Compojure 和 Hiccup。

我们：你觉得 Ring 在 Web 应用编程方面的表现如何？

James: Ring 基于一个单一、简单的抽象，这样的方式使程序员只需要使用函数和 Clojure 的数据结构就可以编写 Web 应用。这意味着 Ring 发挥了 Clojure 的强大特性，使用小且独立的函数让开发更大型的应用更容易。

Ring 可以产生性能上的提升，运行起来只比 Java 或 Go 这样具有静态类型的语言慢一点，但是会比解释型语言例如 Ruby 和 Python 更快。Ring 更常见于构建 REST 风格的 Web 服务，或许是因为通过 Ring 可以更加容易地抽取出可复用的公共功能。

我们：对于 Web 来说，Ring 的哪部分你最喜欢？

James: Ring 让我仅通过函数和 Map 就可以操作 HTTP，而不是其他的类或者类型。由于 Ring 可以很好地发挥语言自身提供的工具，所以你不会遇到太多的困难。我发现在 Ring 里越来越多地使用到了高阶函数（例如中间件），因为高阶函数很容易被组合在一起。

我们：和可定制化的库集合比较，你觉得全功能栈（Full-stack）的框架（例如 Rails 或者 Django）如何？

James: 全功能栈的框架填补了那些无须使用框架的小型应用和框架难以支撑的大型应用的空白, 在复杂性和开发时间上找到了一个很好的平衡点。

因此我认为, 全功能栈框架会承受来自两方面的压力。一方面, 开发者开始喜欢通过更小的应用服务来组织分布式系统, 这样可以借助更多轻量级的库; 另一方面, 函数库变得越来越好, 我们不需要框架提供的项目骨架就可以构建项目。

我推测 Rails 和 Django 这样的框架仍然会有一席之地, 至少在可见的未来是如此。但我猜它们未来的应用场景会减少, 或者以更先进的方式被使用。

4.5 总结

Ring 是一个针对 HTTP 请求和响应的简单数据抽象, 正是这样的简单性让它更强大。把 HTTP、SQL、HTML 和路由信息都转变为数据意味着你可以使用 Clojure 中强大的工具操作和组合它们, 从而实现你的想法。每一个库就像一类乐高积木, 每一个 Ring 应用都是这样一些简单组件的唯一组合。而 Ring 的数据把所有东西连接到了一起。

我们把注意力集中在传统 Web 应用架构上, 以此展示 Ring 以及相关函数库的用法, 这样就不会让你迷失在新的语言和框架中。这些函数库以多种可替代架构存在, 包括集成不同的队列系统 WebSockets, 甚至是大型 Web 应用中能遇到的所有其他基础组件。

4.5.1 Ring 的强项

由于 Ring 是用 Clojure 编写的, 所以其很多能力直接来自于 Clojure。Lisp 被看作是最强大的编程语言, 而 Clojure 继承了 Lisp 大部分强大的功能。虽然 Clojure 并不像 Lisp 那样强大, 但这是使用高级的工具解决现代编程问题的实际案例。

Java 相关技术的生态圈很丰富, Java 的代码可以解决几乎你能想象到的每一个问题。它被作为 Web 应用, 特别是企业级 Web 应用的首选。JVM 令人印象深刻的工程实践和几乎不可撼动的性能优势, 仅次于原生语言如 C 或 C++。

由于 Clojure 基于 Java 平台和 JVM, 所以和 Java 代码集成是很简单的。用 Clojure

实现 Clojure 的动态特性要比用 Java 实现更加简单直接。Clojure 代码可以直接调用 Java 代码, Java 代码也可以直接调用 Clojure 代码。很多实际上的功能会让和 Java 一起工作更简单, 例如 Clojure 的集合都实现标准 java collection 接口。

Ring 和我们看到的工具库具有很多类似集成方面的优势。Ring 建立在 Java Servlet 和 Jetty 之上, Enlive 使用 Tag Soup 库, 都是使用 Java 编写的。Korma 基于 Java 的数据库连接基础架构构建。你无须为库创建新的应用, 因为能想象到的功能已经存在于 Java 中了。

在 Clojure 以操作数据的方式工作, 相较于其他你能找到的语言更加简单。它的语法不仅仅支持 list 和 map 的操作, 同时也支持 set 和 vector。线程操作符让链式转化处理看起来就像 Unix 命令管道。

随着把 HTTP、CSS 选择器、HTML 元素甚至是程序代码转化为数据, Clojure 的神奇工具箱可以把这些问题一并解决。当使用 Clojure 一段时间之后, 这个强大的概念就会成为你的本能。Ring 和其他相关的项目可以让这些扩展用法产生更大的威力。

函数式编程语言是关于组合的研究, 函数作为一等公民可以被作为值传递、返回、赋值。这使你能够把函数组合在一起构建出高层, 抽象地去解决问题。可能你在类似于 Javascript 和 Ruby 这样的语言中尝到了甜头, 而这些能力在 Clojure 中也已经非常成熟了。

Ring 中间件就是组合的绝佳例子。每一个中间件函数都做简单的事情(把字符串的 key 转化为关键词或者解析查询和表单参数), 所有这些 Web 请求都可以通过构建不同的函数组合来处理。Compojure 使用组合在层之间绑定路由规则。

组合是一个和面向对象编程继承完全不一样的工具。与继承不同的是, 你会发现组合对于 Web 应用来说更加自然和易于处理, 因为 Web 应用本身就是各种资源的组合。通过 Ring 实现的应用就是自己的组合, 每部分都通过组合的方式实现。

4.5.2 Ring 的弱项

所有的框架、库和语言都有自己的弊病, 没有哪种工具是完美的。通过 Ring 实现的应用在许多方面有优势, 但是作为刚踏上编写 Ring 应用之旅的你需要始终注意

一些事情。

虽然 Ring 和 Clojure 在访问和集成 Java 方面是一个巨大的优势，但你仍然需要熟悉 Java 才可以发挥它的优势。很多曾经使用 Ruby 和 Python 的 Web 开发人员已经远离 Java 很长时间，他们不知道什么方式是更合理的，而掌握 Java 知识是发挥 Clojure 全部潜力的关键。如果你是个 Java 和 Clojure 的新手，它的学习曲线可能会有点陡。

大多数 Ring 平台是定制化的，你可以选择自己喜欢的组件解决问题。虽然在 Clojure 中存在全功能栈的框架，但大多数平台都只解决一个小问题的函数库，这些函数库与其余函数库能够协同工作。Ring 的数据抽象就是负责处理这些函数库之间的相互操作。

这意味着你可以创建自定义解决方案，但你只需支付使用的那部分组件的开销和性能。这也意味着你必须了解这些组件，以及将这些组件组合在一起的最佳方式。在你找到满足自己需求的工具之前，可能不得不在不同的工具之间进行尝试。由于有太多的组合方式，所以这些组合起来的工具可能缺乏文档。

4.5.3 最后的思考

Clojure 借助 Lisp 的力量解决 Web 问题，Ring 同样使用 Clojure 的能力运行应用。你可以把 Web 应用看作是一个数据转化的集合，在这种视角下，Clojure 确实提供了非常丰富的高级工具让你的工作更加轻松。Clojure 的动态特性会让 Ruby、Javascript 和 Python 程序员感觉很熟悉，无缝地访问 Java 平台不会让你的老板不满。就像你有一堆乐高积木，虽然你可以通过无限的组合创建任何东西，但是也同样需要一些创造力和想象力。

第 5 章

Webmachine

Webmachine 和你之前看到过的其他框架完全不一样，有点像苏格拉底教学法或者 ChooseYourOwnAdventure（译注：“选择你自己的冒险”，又译作“惊险岔路口”。它是一个儿童书籍系列，每一个故事都以第二人称的口吻写成，来使读者能体验主要角色。根据读者的选择，情节也会以不同的形式展开，并导致不同的结局。）Webmachine 会向你的应用询问问题，而答案则决定了通过 HTTP 决策树的路径。

这个 URL 存在吗？如果不存在，返回 404；如果存在，则继续。这个 URL 支持什么样的内容？它支持什么样的编码？它最后一次是什么时候修改的？对于每一个问题，你只需要提供简单的答案，Webmachine 就能把这些答案转化为复杂的 HTTP 请求处理过程。这让其他系统无法比拟。

Erlang 久经考验的可靠性和对并发的良好支持为这一切提供了坚实的基础。其他语言的框架正在从 Erlang 和 Webmachine 这种强大的组合中汲取灵感。

5.1 Webmachine 简介

HTTP 通常被认为是简单的协议，但是实际上非常复杂。大多数框架仅仅暴露出 HTTP 协议最基础的部分，例如根据不同的 HTTP 方法进行路由并且返回不同类型的内容。这些框架都隐藏或忽略了其他功能，例如内容协商（content negotiation）、编码（encoding）和缓存（caching）。

图 5-1 简单的 HTTP 处理流程，展示了大部分 Web 框架是怎样暴露 HTTP 协议的。

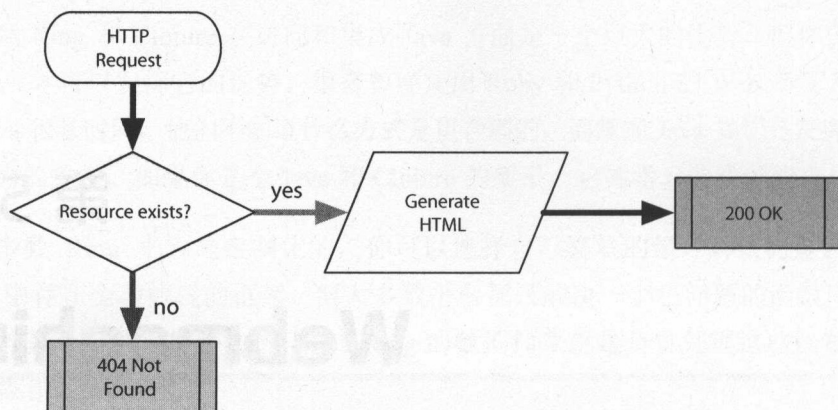


图 5-1 简单的 HTTP 处理流程

通过图 5-1 很容易理解 HTTP 的请求逻辑，但是它隐藏了 HTTP 处理过程的大部分内容。如果你想要缓存或者内容协商，就必须自己在控制器里实现决策逻辑。有时候其他框架提供一些扩展点来处理这些 HTTP 逻辑，例如 Ring 中间件。但这个中间件功能很有限，只能够修改 HTTP 请求和响应数据，你的资源必须始终检查中间件的数据才能做出决策。

Webmachine 更加完整地暴露了 HTTP 协议。图 5-2 应该可以让你认识到 HTTP 有多复杂。在这里每一个菱形都是一个判断，Webmachine 会向你的代码询问一个问题，加粗的箭头和深色的菱形展示了一个 HTTP 请求返回 200 OK 响应的普通流程。你会注意到这个普通的流程都需要经过多次判断。

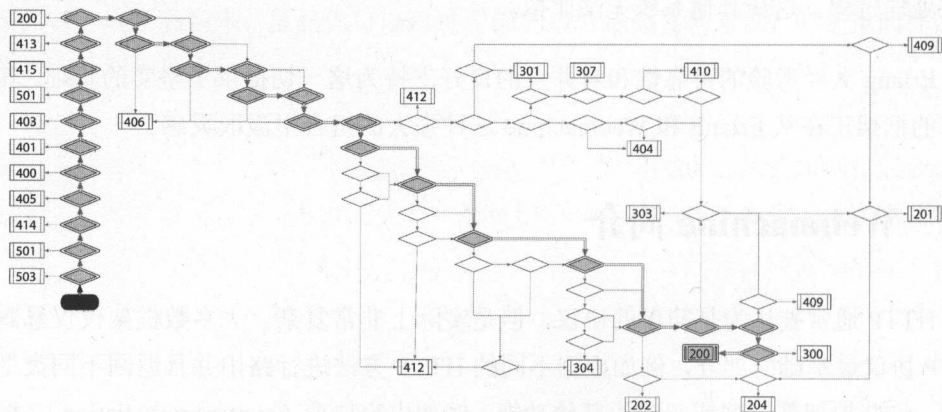


图 5-2 HTTP 处理流程

大多数 Web 应用程序不需要做这么复杂的判断, 因为程序员或者代码中已经通过预先设定答案隐藏了这些细节。Webmachine 暴露了所有的判断并提供了完整的控制, 通过优选的默认答案简化了整个控制流程。如果你的应用需要对某一个判断实现一些不同的行为, 它也提供对应的机制。

这个目标的一个优点是把决策逻辑和生成独立答案的代码分离, 另外一个优点是你不再需要记住这些 HTTP 的状态代码。301 和 302 有什么区别? 你应该返回 401 还是 403? Webmachine 会提供正确的响应来回答这些问题。

在第一天里, 我们将通过第一个 Webmachine 应用探索 Webmachine 的资源函数。接下来, 我们会看到 Webmachine 是如何分发 HTTP 请求的。

在第二天里, 我们将实现 Petite 的第一个版本。这是一个缩短链接的应用, 将利用 Webmachine 提供的额外 HTTP 控制能力。当然, 我们也会用到模板和内容协商。

第三天的内容全部都是关于缓存控制和用户鉴权, 它们是现代 Web 应用中非常重要的部分, 但很多框架都忽视了这两部分。让我们开始学习吧!

5.2 第 1 天: HTTP 请求状态机

Webmachine 善于把复杂的 HTTP 处理变得简单, 作为 Web 系统的后端有非常多的好处。它常被用来创建 API, 例如 Opscode 的 Chef Server API 或者是像 Bsho Riak 数据库这样的 HTTP 接口分发系统。我们将通过缩短链接的服务来了解 Webmachine。当今天的内容结束的时候, 你会准备好创建自己的缩短链接服务。

缩短链接服务从 2002 年第一次出现之后不断激增¹, 把冗长的 URL 压缩为字母数字的序列可以适应有限的显示空间以及减少需要的输入。它的工作原理就是在一张很大的表里查找到目标链接并且进行重定向。这并不符合 Web 应用程序的正常流程, 大多数框架则是改变 HTTP 的流程, 而不是转换应用程序的状态。但这对 Webmachine 来说, 则轻而易举。

¹ http://en.wikipedia.org/wiki/URL_shortening

5.2.1 起步

Webmachine 构建在 Erlang 之上。Erlang 是一个强壮的、并发的函数式编程语言，原本用于编写通信行业的应用程序。你需要安装 Erlang 运行时的环境和一些开发库，完整的打包版本可以在 Linux 包管理器或者 Mac OS X 中的 Homebrew 里找到。Windows 的二进制安装包可以在 Erlang 的官方站点 (<http://www.erlang.org>) 上下载到。近期版本的 Erlang 都很稳定。当安装了 Erlang 以后，你可以通过启动 Erlang 的命令行检查所有的组件是否工作。它应该会显示出当前 Erlang 的版本号和 Erlang 提示符。

```
$ erl
Erlang R15B03 (erts-5.9.3.1) [source] [64-bit] [smp:8:8] [async-threads:0]
[hipe] [kernel-poll:false] [dtrace]

Eshell V5.9.3.1 (abort with ^G)
1>
```

你可以两次按下[Ctrl-C]组合键退出 Erlang 的命令行。接下来，你需要从 Github 下载 Webmachine。最简单的办法是使用 Git 克隆 Webmachine 的代码仓库，且不需要额外安装在特别的路径下。它会通过自带的 Webmachine 项目在指定的位置为我们创建一个新的项目。在你指定的目录里执行以下命令：

```
$ git clone https://github.com/basho/webmachine.git
Cloning into 'webmachine'...
remote: Counting objects: 2542, done.
remote: Compressing objects: 100% (1291/1291), done.
remote: Total 2542 (delta 1468), reused 2247 (delta 1210)
Receiving objects: 100% (2542/2542), 1.80 MiB | 39 KiB/s, done.
Resolving deltas: 100% (1468/1468), done.
```

所有准备都已就绪，我们可以开始创建 Webmachine 应用了。你不用像其他框架那样需要安装和配置 Webmachine，Erlang 的虚拟机和 Rebar 包管理器会让你的起步变得很轻松。

5.2.2 Hello, World

Webmachine 通过一个叫作 new_Webmachine.sh 的脚本创建一个新的应用，包括一些简单的资源、基本路由以及构建脚本。只需要给它传入你的项目名称和项目路径，

它就会在指定位置创建应用。项目路径是可选参数:

```
$ webmachine/scripts/new_webmachine.sh hello
==> priv (create)
Writing /Users/jack/src/hello/README
Writing /Users/jack/src/hello/Makefile
Writing /Users/jack/src/hello/rebar.config
Writing /Users/jack/src/hello/rebar
Writing /Users/jack/src/hello/start.sh
Writing /Users/jack/src/hello/src/hello.app.src
Writing /Users/jack/src/hello/src/hello.erl
Writing /Users/jack/src/hello/src/hello_app.erl
Writing /Users/jack/src/hello/src/hello_sup.erl
Writing /Users/jack/src/hello/src/hello_resource.erl
Writing /Users/jack/src/hello/priv/dispatch.conf
```

这些文件包括大多数 Erlang 应用都会有的内容和一些简单的占位文件 (placeholders)。只需要运行 `start.sh` 就会启动应用。`src/hello_resource.erl` 和 `priv/dispatch.conf` 分别定义了主要的资源和 URL 路由。

我们可以通过执行 `make` 和 `start.sh` 构建并运行这个应用。默认情况下, 应用会监听 8000 端口, 所以你需要在浏览器里指明端口号。在浏览器里输入 `http://localhost:8000/`, 就可以看到 Webmachine 的欢迎信息。

```
$ make
==> hello (get-deps)
Pulling webmachine from {git,"git://github.com/basho/webmachine","HEAD"}
Cloning into 'webmachine'...
<<omitted output>>
==> hello (compile)
Compiled src/hello_app.erl
Compiled src/hello_resource.erl
Compiled src/hello.erl
Compiled src/hello_sup.erl

$ ./start.sh
Erlang R15B03 (erts-5.9.3.1) [source] [64-bit] [smp:8:8] [async-threads:0]
[hipe] [kernel-poll:false] [dtrace]
<<omitted output>>
=PROGRESS REPORT==== 2-May-2013::22:52:15 ===
    application: hello
    started_at: nonode@nohost
```

你可以连续两次按下 [Ctrl-C] 组合键, 以退出 Erlang 的虚拟机。让我们先来看看两个最重要的文件: `dispatch.conf` 和 `hello_resource.erl`。`dispatch.conf` 文件是一个 Erlang 的路由项序列。每一个路由项以点号结束, 这些转发规则将 URL 指向不同的资源。

```
webmachine/hello/priv/dispatch.conf
{[], hello_resource, []}.
```

这个文件里只有一条简单的规则，包含 3 个部分：一个路径定义、一个资源模块以及资源模块的参数。这里的空列表“[]”表示根路径。`hello_resource` 就是资源模块，它定义在一个同名的`.erl`文件里。由于实现 Webmachine 的资源不需要参数，所以紧随其后的[]是一个空的参数列表。这个文件非常简单，我们随后会看到一些复杂的转发规则。

`hello_resource.erl` 定义了一个叫作 `hello_resource` 的 Erlang 模块，实现了一个资源。每一个导出的资源函数都被 Webmachine 的状态机用来回答问题，从而决定 HTTP 请求如何被处理。

```
webmachine/hello/src/hello_resource.erl
-module(hello_resource).
① -export([init/1, to_html/2]).
② -include_lib("webmachine/include/webmachine.hrl").

③ init([]) -> {ok, undefined}.

④ to_html(ReqData, State) ->
    {"<html><body>Hello, new world</body></html>", ReqData, State}.
```

① 注意这些资源函数，它们都是以/1 或者/2 结尾的，是 Erlang 的函数签名。在 Erlang 里，名字相同但是参数数量不同则视为不同的函数。`init` 在资源初始化的时候被调用，转发规则就是传给它的参数。`to_html` 是默认的内容函数，Webmachine 会调用它去回答应该给用户展现什么样的内容。接下来我们将看到很多资源函数。

② 所有的 Webmachine 资源必须包含 `Webmachine.hrl`，它定义了 HTTP 请求的数据结构。

③ `init` 函数为 HTTP 请求设置资源状态，这个状态会连续传递给每个资源函数。例如，它作为第二个参数传递给了 `to_html`。由于这个资源没有内部状态，所以返回 `undefined`。数据在 Erlang 里是不可改变的，所以函数基本上带着原来的状态并返回一个新状态，然后传递给下一个函数。这些无状态的流程，让函数式的程序很容易做出推理。

④ 资源函数返回一个三元组作为结果，请求数据和新的内部状态。内容函数的结果经常是一个 `iolist`，它可能是一个字符串、一个二进制或者一个 `iolists` 的列表。

就像你所看到的，它只需要一点基础设施就可以让 Webmachine 应用运行起来。

几乎没有什么资源模块和分发请求,当然这只是开始。当我们开始向自己的资源模块里增加资源函数后,真正有意思的部分便开始了。

5.2.3 和资源函数一起工作

Webmachine 资源可以使用超过 30 个资源函数定制 HTTP 请求处理的行为。Webmachine 通过调用一个资源函数提出一个问题,你的资源会提供一个简单的答案,通常是 true 或者 false。你的资源模块不需要实现每个函数,因为 Webmachine 为它们提供了默认的行为。所有的资源函数以及它们的默认行为和合法的答案,全部都在文档里。最基本的资源函数是 `content_types_provided`,它回答给 Webmachine 你的资源展现什么样的内容或者什么内容类型的问题。默认情况下,Webmachine 假设一个简单的 HTML 展现定义在了一个叫作 `to_html` 的函数里。但当实现 `content_types_provide` 的时候,你可以添加替换它的内容,或者通过 Webmachine 调用其他的函数提供内容。

下一个例子展示了如何让你返回同一个内容资源的 HTML 和纯文本两种形式。你只需要实现 `content_types_provide` 的 `to_html` 和 `to_text` 这两个资源函数就可以做到。

```
webmachine/hello2/src/hello2_resource.erl
-module(hello2_resource).
-export([init/1,
         content_types_provided/2,
         to_html/2,
         to_text/2]).
```

```
-include_lib("webmachine/include/webmachine.hrl").
```

```
init([]) -> {ok, undefined}.
```

- ① `content_types_provided(ReqData, State) ->`
`{[{{"text/html", to_html},`
 `{"text/plain", to_text}], ReqData, State}.`
`to_html(ReqData, State) ->`
`{ "<html><body>Hello, HTML world</body></html>\n", ReqData, State}.`
- ② `to_text(ReqData, State) ->`
`{"Hello, text world\n", ReqData, State}.`

① 这里除了 `to_html` 以外还增加了一个处理函数 `to_text`, 用于返回 `text/plain` 类型的内容。

- ② 这个处理函数除了返回的是文本，剩下的几乎和 `to_html` 相同。

这个简单的改动让我们的小应用可以根据 HTTP 客户端请求的 `Accept` 报头改变它的行为。客户端会通过 `Accept` 报头传入一个它可以处理的媒体类型列表。`Webmachine` 会利用这个信息选择合适的内容处理器。注意你无须实现内容协商，只需要回答 `Webmachine` 什么样的内容可用。我们可以用 `curl` 在命令行里模仿这样的行为：

```
$ curl --header 'accept: text/html' http://localhost:8000/
<html><body>Hello, HTML world</body></html>

$ curl --header 'accept: text/plain' http://localhost:8000/
Hello, text world
```

大多数框架不会暴露这些信息，这也是为什么你会看到很多 API 的编码格式选择会放在 URL 中。例如，Twitter 的开放 API 返回内容的格式基于 URL 的后缀名。`statuses/public_timeline.json` 会返回 JSON 格式的数据，`statuses/public_timeline.atom` 则会返回一个 Atom 的订阅(Atom feed)，这样可以避免滥用 HTTP 协议。这两个 Twitter 的 API 访问点，实际上是对同一个内容返回不同的格式。在 `Webmachine` 里，这个结构是预留的。

5.2.4 资源函数

让我们看一些对缩短链接非常有用的资源函数。我们的链接缩短应用事实上不包含用户最终想要的内容。`Webmachine` 通过调用 `resource_exists` 函数询问一个资源是否可用。一般情况下，`Webmachine` 假设是 `true`，但是在这种短链接的情况下肯定不会。

在 `hello2` 项目创建 `src/uncertain_resource.erl` 文件，它仅仅是对原来的“HelloWorld”例子的细微改动：

```
webmachine/hello2/src/uncertain_resource.erl
-module(uncertain_resource).
-export([init/1,
         to_html/2]).

-include_lib("webmachine/include/webmachine.hrl").
init([]) -> {ok, undefined}.
to_html(ReqData, State) ->
    {"nothing to see here", ReqData, State}.
```

你需要修改 `priv/dispatch.conf` 为新的资源添加转发规则:

```
webmachine/hello2/priv/dispatch.conf
{[], hello2_resource, []}.
[{"uncertain"}, uncertain_resource, []}.
```

编译和运行 `hello2`, 当我们访问 `http://localhost:8000/uncertain` 时应该和你期望的一样。现在, 让我们实现 `resource_exists`。它非常简单, 只有两个可能的答案:

```
webmachine/hello2/src/uncertain_resource.erl
%% remember to add resource_exists/2 to the export list

resource_exists(ReqData, State) ->
    {false, ReqData, State}.
```

重新编译并打开 URL, 你现在应该得到一个 404 Not Found 的错误。这个正是当资源不存在时应该发生的。但是注意我们既没有修改资源的内容函数, 也没有在任何地方指定什么请求应该返回 404, 只是回答了一个简单的问题, Webmachine 就能处理好剩下的细节。

Webmachine 就是以这样的方式, 不断向我们的资源提出问题。试试为 `previously_existed` 添加一个实现:

```
webmachine/hello2/src/uncertain_resource.erl
%% remember to add previously_existed/2 to the export list

previously_existed(ReqData, State) ->
    {true, ReqData, State}.
```

让我们看看 Webmachine 现在是怎么做的:

```
$ curl -i http://localhost:8000/uncertain
HTTP/1.1 410 Gone
Server: MochiWeb/1.1 WebMachine/1.10.0
Date: Sun, 12 May 2013 04:12:09 GMT
Content-Type: text/html
Content-Length: 0
```

你知道在这个例子里的 HTTP 状态编码吗? 你从没有在应用里用到状态编码 410, 但是 Webmachine 总是知道如何选择正确的 HTTP 状态编码。因为它就是基于整个 HTTP 状态机建模的。

如果资源当前不存在, 可它曾经存在, 它去哪里了? Webmachine 通过调用 `moved_permanently` 和 `moved_temporarily` 进行查询。`moved_permanently` 更适合短链接服务,

让我们来实现 `moved_permanently` 然后看看 Webmachine 会怎么做。

```
webmachine/hello2/src/uncertain_resource.erl
%% remember to add moved_permanently/2 to the export list

moved_permanently(ReqData, State) ->
    {{true, "http://pragprog.com/"}, ReqData, State}.

$ curl -i http://localhost:8000/uncertain
HTTP/1.1 301 Moved Permanently
Server: MochiWeb/1.1 WebMachine/1.10.0
Location: http://pragprog.com/
Date: Sun, 12 May 2013 04:25:36 GMT
Content-Type: text/html
Content-Length: 0
```

Webmachine 再一次准确地推断出了要做的事情。Webmachine 状态机的决策过程可参看图 5-3 Webmachine 存在状态机逻辑。其他的资源函数的工作方式非常相似，但有一些比 `true` 和 `false` 更复杂的答案。我们将在后面的部分了解这些内容，但现在来讨论一下请求转发。

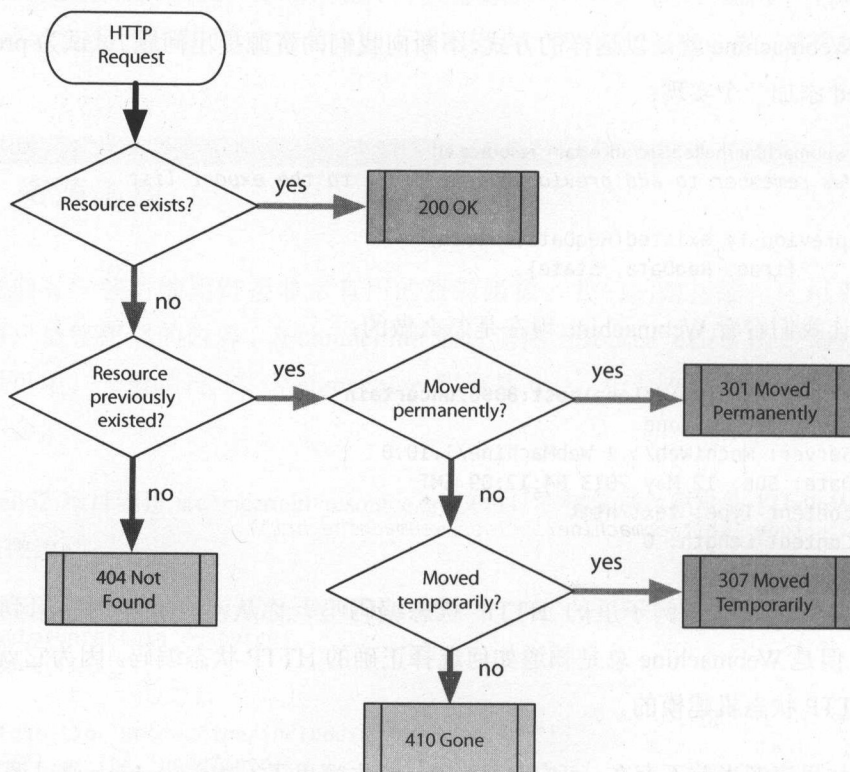


图 5-3 Webmachine 存在状态机逻辑

5.2.5 请求转发

请求转发是 Webmachine 根据请求选择资源的过程。你可能想把所有以/user 结尾的 URL 发送给同一个资源, 或者根据是否是单个数据或数据集切换不同的资源。几乎任何逻辑都可以实现, 但是 Webmachine 让这些通用的模式更加简单, 让我们把注意力集中在些这上面。

创建一个新的 Webmachine 项目叫作 dispatcher, 然后看看 priv/dispatch.conf 文件:

```
webmachine/dispatcher/priv/dispatch.conf
{[], dispatcher_resource, []}.
```

之前已经看到了这个分发规则, 现在让我们把它分解以便更细致地了解它。

每一条规则都是一个三元组或者四元组。对于三元组来说, 第一个元素是路径定义, 第二个元素是匹配这条规则的 Erlang 资源模块, 最后一个元素是传给资源模块 init 函数的参数。

这个路径的定义是[], 它实际上是根目录/, 每一项在这个列表里都是一个路径项, Webmachine 会使用斜线把请求路径分解成片段, 然后分别匹配每一个片段。例如, [“hello”, “world”] 将会匹配/hello/world。

一个资源可以使用 wrq:path(ReqData) 函数解析路径。修改 src/dispatcher_resource.erl, 让它可以打印路径, 添加 [“hello”, “world”] 转发规则, 然后构建并运行 dispatcher 应用。

```
webmachine/dispatcher/src/dispatcher_resource.erl
-module(dispatcher_resource).
-export([init/1, to_html/2]).
-include_lib("webmachine/include/webmachine.hrl").
```

```
init([]) -> {ok, undefined}.
to_html(ReqData, State) ->
    [{"you asked for ", wrq:path(ReqData), "\n"},
     ReqData, State}.
```

```
webmachine/dispatcher/priv/dispatch.conf
[["hello", "world"], dispatcher_resource, []].
```

```
$ curl http://localhost:8000/hello/world
you asked for /hello/world
```


Webmachine 不限于特定路径的标记,也可以使用其他的符号作为路径标记来调用资源函数。

5.2.6 参数化转发

Webmachine 有一个特别的路径条目,可以匹配任意数量的路径条目: '*'. 注意这是 Erlang 的原子*,不是一个字符串,原子如果不以小写字母开头就必须在单引号里。这个路径条目只能在路径定义的最后出现。匹配*条目的那部分路径可以通过调用 `wrq:disp_path(ReqData)` 检索。创建一个新的模块 `star_resource`,用于打印出转发路径,然后为[“hello”, '*']添加一个转发规则。让我们重新构建和运行 dispatcher:

```
webmachine/dispatcher/src/star_resource.erl
-module(star_resource).
-export([init/1, to_html/2]).

-include_lib("webmachine/include/webmachine.hrl").

init([]) -> {ok, undefined}.

to_html(ReqData, State) ->
    [{"you asked for ", wrq:path(ReqData), "\n",
      "star path was ", wrq:disp_path(ReqData), "\n"},
     ReqData, State].
```

```
webmachine/dispatcher/priv/dispatch.conf
[["hello", '*'], star_resource, []].

$ curl http://localhost:8000/hello/webmachine
you asked for /hello/webmachine
star path was webmachine

$ curl http://localhost:8000/hello
you asked for /hello
star path was

$ curl http://localhost:8000/hello/how/are/you
you asked for /hello/how/are/you
star path was how/are/you
```

Webmachine 也可以让你绑定路径条目和命名的原子。一个[“goodbye”, who]的路径定义匹配任意一个有两个部分且第一个部分是 goodbye 的条目。除此之外,

`wrq:path_info(who, ReqData)`将会根据绑定名称返回第二个路径条目。让我们添加一个 `named_resource` 的条目和一个合适的转发规则。

```
webmachine/dispatcher/src/named_resource.erl
-module(named_resource).
-export([init/1, to_html/2]).

-include_lib("webmachine/include/webmachine.hrl").

init([]) -> {ok, undefined}.

to_html(ReqData, State) ->
    {[ "goodbye, ", wrq:path_info(who, ReqData), "\n" ],
      ReqData, State}.
```

```
webmachine/dispatcher/priv/dispatch.conf
[["goodbye", who], named_resource, []].

$ curl http://localhost:8000/goodbye/world
goodbye, world
```

如果 `wrq:path_info(ReqData)` 没有指定命名, 将会返回命名条目的全部属性列表。

5.2.7 我们在第 1 天学到的

Webmachine 这种通过 HTTP 状态机模型处理 HTTP 请求不仅独特, 而且使用 Erlang 实现, 这个函数式的编程语言可能与你之前使用的大多数其他的语言完全不同。你可能对它比较陌生, 但从 Webmachine 处理问题的方式中我们仍然能得到一些启发。

用 Webmachine 编写应用就像为 HTTP 请求编写一本自己的《惊险岔路口》。Webmachine 询问一些简单的问题, 你提供简单的答案, Webmachine 就可以指引复杂的 HTTP 状态机帮你完成 HTTP 处理。其本质是引导对 HTTP 协议的访问。而其他的框架则把这些细节隐藏起来, 变得难以交互。

第 1 天的自学

查阅

- 在 Internet 上查找基于 Webmachine 应用的例子。
- Webmachine wiki。

实践

- 找出如何在你的资源函数里处理查询字符串参数（提示：探索 wrq 模块中函数的可用性，可以在 Webmachine 的 Wiki 里查看 Request Data API）。
- 在 Webmachine Wiki 里阅读 Mechanics 部分了解它整体是如何工作的。

5.3 第2天：构建应用

昨天我们看到缩短链接服务需要的大部分片段，今天我们会把这些片段拼接起来完成这个应用程序的第一个版本——Petite。

今天我们会看到 Webmachine 中一些基本的前端任务，并会用相关的前端库为 Petite 构建 UI。你将看见如何用 `mustache.erl`¹ 把 HTML 模板集成到一个 Webmachine 资源里，它是 Mastache 引擎的 Erlang 实现。

Webmachine 可以很容易地处理不同类型的输入数据，以便在相同的资源上进行 Web 交互和 API 交互。你会发现，输入数据的展现类型和资源本身的展现形式是 Webmachine 的重要组成部分。

5.3.1 短链接

我们将应用昨天学到的东西来构建 Petite 的第一个迭代。这个链接缩短应用的一个版本可以缩短链接，并把用户的请求重定向到真实的 URL。

首先，新建一个叫作 Petite 的 Webmachine 项目。从这个基本的雏形开始，我们将持续扩展 Petite。

压缩和存储

在我们为 Petite 的 API 编写 Webmachine 资源模块之前，必须：①找到一个缩短链接的方法；②构建一个短链接和真实链接相互映射的查找表。幸运的是，Erlang 内置了解决这两个问题的工具。

¹ <https://github.com/mojombo/mustache.erl>

如果你想让一个数字字符串更短，最简单的办法就是利用一个更大的进制表进行转化。比如二进制数 10000000 在 10 进制就是 128，在基数为 36 时就是 3K（基数为 36 就是利用 26 个英文字母+10 个阿拉伯数字作为进制编码）。利用这个办法，我们让每一个数值对应一个真实的 URL，这个数值会自动增长。返回的编码只是一个用高进制表示的数值，这样可以使它尽可能地紧凑。Erlang 可以通过 `integer_to_list (Number,Base)` 将整数转化为以 36 为基数的列表。

存储链接查找表可以有好几种方法，最简单的是使用一个 ETS (Erlang Term Storage) 表，它是 Erlang 标准库中的一部分。一个 ETS 表就是一个内存中的键值对。你可以通过 Erlang 的元组进行存取，元组的第一个元素是键，整个元组是值。

把这两个部分组合到一起，我们可以编写一个 `gen_server` 模块。它是一个 Erlang 服务，用来为给定的 URL 生成短代码，同时能根据短代码返回 URL。由于篇幅的关系，这里对 `gen_server` 就不详细介绍了。如果你感兴趣，可以看看 Francesco Cesarini 和 Simon Thompson 联合撰写的 Erlang Programming[CT09]，或 Joe Armstrong 的书 Programming Erlang [Arm13]。让我们看看重要的部分：

```
webmachine/petite/day1/petite/src/petite_url_srv.erl
```

```
-module(petite_url_srv).
```

```
%% public API
```

```
-export([start_link/0,  
        get_url/1,  
        put_url/1]).
```

```
-behaviour(gen_server).
```

```
-export([init/1,  
        terminate/2,  
        code_change/3,  
        handle_call/3,  
        handle_cast/2,  
        handle_info/2]).
```

```
-define(SERVER, ?MODULE).
```

```
-define(TAB, petite_urls).
```

```
-record(st, {next}).
```

```
%% public API implementation
```



```

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

① get_url(Id) ->
    gen_server:call(?SERVER, {get_url, Id}).
put_url(Url) ->
    gen_server:call(?SERVER, {put_url, Url}).

%% gen_server implementation

② init(_) ->
    ets:new(?TAB, [set, named_table, protected]),
    {ok, #st{next=0}}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

③ handle_call({get_url, Id}, _From, State) ->
    Reply = case ets:lookup(?TAB, Id) of
        [] ->
            {error, not_found};
        [{Id, Url}] ->
            {ok, Url}
    end,
    {reply, Reply, State};

④ handle_call({put_url, Url}, _From, State = #st{next=N}) ->
    Id = b36_encode(N),
    ets:insert(?TAB, {Id, Url}),
    {reply, {ok, Id}, State#st{next=N+1}};

handle_call(_Request, _From, State) ->
    {stop, unknown_call, State}.

handle_cast(_Request, State) ->
    {stop, unknown_cast, State}.

handle_info(_Info, State) ->
    {stop, unknown_info, State}.

%% internal functions

⑤ b36_encode(N) ->
    integer_to_list(N, 36).

```

① 这个模块的公共 API 简单地把任务委托给了服务器进程。这在 `gen_serve` 的实现方式中很普遍，因为 `server` 自身可能会改动内部消息的格式。

② 我们通过创建一个新的 ETS 初始化了 `server`，这个 ETS 用来存储短链接代码和对应的 URL。我们的计数器的初始值是 0。Webmachine 通过 `State` 变量把资源 `init` 函数返回的值传递给其他资源函数，包括 `gen_server`。这也是 Webmachine 处理问题的方式。

③ 通过查找 ETS 获得真实 URL 很容易。

④ 把 URL 交给 `server` 去创建一个编码，同时插入一条对应的记录。注意，计数器在返回的 `State` 变量中加 1。

⑤ 创建一个编码就和 `integer_to_list` 一样简单，目前你不需要比 36 更高的进制。

`supervisor` 进程会捕获 `gen_servers` 的状态以确保它在运行，如它崩溃后会重启 `sever`。为了使用 `petite_url_srv`，我们必须把它添加到 `Petite` 的主 `supervisor` 进程，`petite_sup`。下面 `init` 函数里的标准部分就是这些必要的修改。

```
webmachine/petite/day1/petite/src/petite_sup.erl
```

```
Web = {webmachine_mochiweb,
        {webmachine_mochiweb, start, [WebConfig]},
        permanent, 5000, worker, [mochiweb_socket_server]},
> UrlServer = {petite_url_srv,
>               {petite_url_srv, start_link, []},
>               permanent, 5000, worker, []},
> Processes = [Web, UrlServer],
{ok, { {one_for_one, 10, 10}, Processes} }.
```

编译并启动 `Petite`，让我们在 Erlang 的命令行里测试一下新服务。注意如果启动应用后你没有看到 `1>` 提示符，再次按下 `Enter` 键就可以了。

```
$ ./start.sh
<<omitted output>>
=PROGRESS REPORT==== 15-May-2013::21:10:14 ===
        application: petite
        started_at: nonode@nohost
1> whereis(petite_url_srv).
<0.92.0>
2> petite_url_srv:put_url("https://pragprog.com/").
{ok,"0"}
3> petite_url_srv:put_url("https://github.com/basho/webmachine").
{ok,"1"}
4> petite_url_srv:get_url("1").
{ok,"https://github.com/basho/webmachine"}
5> petite_url_srv:get_url("3K").
{error,not_found}
```

我们的服务工作正常，已经准备好使用我们创建的 Webmachine 资源了。

重定向

你已经学会了创建 Webmachine 的资源，同时也知道了如何使用 `resource_exists` 和 `moved_permanently` 这样的资源函数重定向 HTTP 请求，还看到了如何用 `wrq:path_info` 检索并且在转发时把路径标识符绑定到原子上。所有这些仍然通过 `petite_url_srv` 组合到一起。这样，Petite 就可以生成短链接了。

首先，在 `priv/dispatch.conf` 文件里为你的新资源创建一条新规则：

```
webmachine/petite/day1/petite/priv/dispatch.conf
{[code], petite_fetch_resource, []}.
```

然后，创建 `petite_fetch_resource` 模块。尝试修改在第 161 页的重定向例子，在下面的实现中使用 `petite_url_srv`：

```
webmachine/petite/day1/petite/src/petite_fetch_resource.erl
-module(petite_fetch_resource).
-export([init/1,
         to_html/2,
         resource_exists/2,
         previously_existed/2,
         moved_permanently/2]).

-include_lib("webmachine/include/webmachine.hrl").

init([]) ->
    {ok, ""}.

to_html(ReqData, State) ->
    {"", ReqData, State}.

resource_exists(ReqData, State) ->
    {false, ReqData, State}.

previously_existed(ReqData, State) ->
    Code = wrq:path_info(code, ReqData),
    case petite_url_srv:get_url(Code) of
        {ok, Url} ->
            {true, ReqData, Url};
        {error, not_found} ->
            {false, ReqData, State}
    end.

moved_permanently(ReqData, State) ->
    {{true, State}, ReqData, State}.
```

重编译 Petite，然后像之前一样在 Erlang 的命令里添加一些链接。一旦它能够把一些链接缩短，你就可以测试这个资源了：

```
$ curl -i http://localhost:8000/1
HTTP/1.1 301 Moved Permanently
Server: MochiWeb/1.1 WebMachine/1.9.2
Location: https://github.com/basho/webmachine
Date: Thu, 16 May 2013 03:29:09 GMT
Content-Type: text/html
Content-Length: 0
```

我们的链接缩短服务已经可以正常运行了，但是仍然缺少一些功能。我们需要一个 HTTP API 提供这样的服务，只要创建一个新资源：petite_shorten_resource。

缩短链接的 API

把一个链接缩短的 API 很简单。HTTP 的 POST 请求将会包含一个带有被缩短 URL 的表单数据。Petite 将会在响应里返回缩短后的链接。

让我们想想 Webmachine 向我们的资源询问的第一个问题，以及我们应该提供什么样的答案。首先我们需要用 HTTP 的 POST 方法回答 `allowed_methods`。接下来由于响应内容是文本，所以资源必须回答 `content_types_provided` 并且用 `to_text` 函数返回响应。Webmachine 要求我们提供一个生成函数体的函数，虽然在这类情况下这不是必需的。

目前，你看到的都是一些处理 HTTP 的 GET 请求的资源函数。对于 HTTP 的 POST 请求，Webmachine 首先调用 `post_is_create` 决定这个请求是否要创建新的资源。如果回答是 `false`，Webmachine 状态机会委托给 `process_post` 处理；如果答案是 `true`，Webmachine 状态机会遵循状态判断路径。我们不需要在这一章覆盖这些内容。即使 Petite 没有通过调用这个 API 创建新的资源，也会遵循之前的路径。`process_post` 必须解析请求中的表单数据，缩短链接，然后生成合适的响应。让我们看看它是如何做到的：

```
webmachine/petite/day1/petite/src/petite_shorten_resource.erl
-module(petite_shorten_resource).
-export([init/1,
        allowed_methods/2,
        process_post/2,
        content_types_provided/2,
        to_text/2]).

-include_lib("webmachine/include/webmachine.hrl").

init([]) ->
    {ok, undefined}.
```



```

allowed_methods(ReqData, State) ->
    {'POST', ReqData, State}.

content_types_provided(ReqData, State) ->
    {[{"text/plain", to_text}], ReqData, State}.

process_post(ReqData, State) ->
    Host = wrq:get_req_header("host", ReqData),
    Params = mochiweb_util:parse_qs(wrq:req_body(ReqData)),
    Url = proplists:get_value("url", Params),
    {ok, Code} = petite_url_srv:put_url(Url),
    Shortened = "http://" ++ Host ++ "/" ++ Code ++ "\n",
    {true, wrq:set_resp_body(Shortened, ReqData), State}.

to_text(ReqData, State) ->
    {"", ReqData, State}.

```

`Wrq:get_req_header` 返回请求报文头部的内容。这里用作取回客户端连接的主机名和端口。我们可以使用这些信息构建短链接。

`mochiWeb_util:parse_qs` 是一个用来解析表单或查询字符串的函数，是由 MochiWeb 提供的。MochiWeb 是一个 HTTP 处理库，Webmachine 和大多数其他用 Erlang 编写 Web 库都是基于它构建的。我们已经为它提供了 `wrq:req_body(ReqData)` 作为输入，它就是从 http 请求中提取出来的内容体。

`mochiWeb_util:parse_qs` 返回一个 Erlang 属性列表，`process_post` 抓取 url 属性，把它传递给内部的链接缩短服务，然后返回新构建好的短链接。

最后，`wrq:set_resp_body` 用来设置短链接响应报文的内容。由于数据在 Erlang 里是无状态的，`wrq:set_resp_body` 修改传进来的 `ReqData` 结构并返回。`process_post` 返回 `true` 用来表明处理成功。

Petite 需要为这个资源制定一个新的转发规则，并放在 `petite_fetch_resource` 规则之前：

```

webmachine/petite/day1/petite/priv/dispatch.conf
[["shorten"], petite_shorten_resource, []].

```

你现在可以重新构建 Petite 并测试它的输出

```

$ curl -i -X POST http://localhost:8000/shorten \
> --data 'url=https%3A%2F%2Fpragprog.com%2F'
HTTP/1.1 200 OK
Server: MochiWeb/1.1 WebMachine/1.9.2
Date: Fri, 17 May 2013 04:56:25 GMT

```

```
Content-Type: text/plain
Content-Length: 24
```

```
http://localhost:8000/0
```

```
$ curl -i http://localhost:8000/0
HTTP/1.1 301 Moved Permanently
Server: MochiWeb/1.1 WebMachine/1.9.2
Location: https://pragprog.com/
Date: Fri, 17 May 2013 04:57:03 GMT
Content-Type: text/html
Content-Length: 0
```

Petite 现在可以把长链接缩短并重定向到原始 URL。开发人员已经用不同的语言和框架实现了这个缩短链接的应用，但是很难再想出比 Webmachine 版本更简单的实现了。把 HTTP 状态作为状态机进行建模，同时用简单的答案分离决策逻辑。Webmachine 把这个问题解决得和编写“Hello, World”一样简单。

即使这个功能非常基础，也可以作为整个 Web 应用内部的短链接服务。当然，你可能会在产品环境中持久化这张查找表。

Petite 还没有一个用户界面。我们来看看 Webmachine 是如何处理前端任务的，然后来解决这个问题。

5.3.2 使用 Mustache 模板引擎

我们的目标是创建一个资源，它可以输出用户最后调用缩短链接服务的原链接。像大多数框架那样，我们将使用模板语言，否则生成 HTML 就变成了一件枯燥乏味的事情。每一种编程语言都有好几种模板库，但是 Mustache 似乎就是为 Erlang 这样的函数语言而生的。

你可能已经在第 2 章 CanJS 看到了 Mustache。或许是因为它的简洁，Mustache 模板系统已经在几乎每一个编程语言里都有实现。Erlang 也不例外，mustache.erl 是 Webmachine 使用的 Mustache 实现。

像大多数模板语言一样，我们需要提供模板本身和模板的上下文。模板可以是字符串或者文件，模板上下文是一个键值对的数据字典，提供了模板渲染时需要访问的变量。模板总是相同的，但上下文会改变。

让我们看一个最简单的例子。创建一个叫作 `template` 的新 Webmachine 项目，编辑 `rebar.config` 文件为我们的项目添加 `mustache` 的依赖：

```
webmachine/template/rebar.config
```

```
{deps, [{webmachine, "1.10.1",
        {git, "git://github.com/basho/webmachine",
          {tag, "1.10.1"}}},
➤ {mustache, "0.1.0",
➤ {git, "git://github.com/mojombo/mustache.erl.git",
➤ {branch, "master"}}}]}.

```

Now make a new resource, `template_basic_resource.erl`, and an appropriate dispatch rule.

```
webmachine/template/priv/dispatch.conf
```

```
{["basic"], template_basic_resource, []}.
```

```
webmachine/template/src/template_basic_resource.erl
```

```
-module(template_basic_resource).
```

```
-export([init/1, to_html/2]).
```

```
-include_lib("webmachine/include/webmachine.hrl").
```

```
init([]) -> {ok, undefined}.
```

```
to_html(ReqData, State) ->
```

```
    Template = "<html><body>Visit {{ url }}</body></html>",
    Context = dict:from_list([{url, "https://pragprog.com/"}]),
    Response = mustache:render(Template, Context),
    {Response, ReqData, State}.
```

模板上下文在这里叫作 `url`，它关联到 Pragmatic 的主页。模板会包含一个同样叫作 `url` 的标签，在你的浏览器里访问 `http://localhost:8000/basic` 会显示“Visit `https://pragprog.com`”。

由于 `Mustache` 模板都是指令，因此展现一个集合的方法都非常相似。集合上下文变量的值应该是一个列表，列表里的每一项在每次迭代中都是新上下文的数据字典。这点在下面这个例子里很容易看到：

```
webmachine/template/priv/dispatch.conf
```

```
{["list"], template_list_resource, []}.
```

```
webmachine/template/src/template_list_resource.erl
```

```
-module(template_list_resource).
```

```
-export([init/1, to_html/2]).
```

```
-include_lib("webmachine/include/webmachine.hrl").
```

```

init([]) -> {ok, undefined}.
to_html(ReqData, State) ->
    Template = "<html><body>" ++
        "<ul>" ++
        "{{#urls}}" ++
        "<li>{{ url }}</li>" ++
        "{{/urls}}" ++
        "</ul>" ++
        "</body></html>",
    ① Urls = [{url, "https://pragprog.com/"},
        {url, "https://github.com/basho/webmachine"},
        {url, "https://github.com/mojombo/mustache.erl"}],
    ② Dicts = [dict:from_list([U] || U <- Urls),
    ③ Context = dict:from_list([urls, Dicts])],
    Response = mustache:render(Template, Context),
    {Response, ReqData, State}.

```

- ① 我们创建了一个表示为元组列表的数据，它在 Erlang 里是属性列表(property lists)。
- ② 我们把通过列表构建新的列表理解为构建数据字典的列表。
- ③ 我们通过把 urls 变量和数据字典列表关联起来创建上下文。

用字符串作为模板有点奇怪，我们可以通过读取模板文件的方式改进：

```

webmachine/template/priv/dispatch.conf
[{"file", template_file_resource, []}].

```

```

webmachine/template/src/template_file_resource.erl
-module(template_file_resource).
-export([init/1, to_html/2]).
-include_lib("webmachine/include/webmachine.hrl").

```

```

init([]) -> {ok, undefined}.
to_html(ReqData, State) ->
    {ok, TemplateBin} = file:read_file(
        code:priv_dir(template) ++ "/simple.mustache"),
    TemplateStr = binary_to_list(TemplateBin),
    Context = dict:from_list([message, "Hello from a file"]),
    Response = mustache:render(TemplateStr, Context),
    {Response, ReqData, State}.

```

```

webmachine/template/priv/simple.mustache
<html>
<body>
    {{ message }}
</body>
</html>

```

code:priv_dir 用于返回 Erlang 应用中 priv 目录的路径，这里就是我们用来存放模板

的地方。Erlang 的应用经常把应用相关的数据放在 `priv` 目录下。这个模板的名字被添加到这个目录里，之后 `file:read_file` 会返回这个文件内容的二进制形式。把它转化为字符串之后，它在 Erlang 里就变成了一个列表，我们可以和之前一样调用 `render` 函数来渲染它。

5.3.3 Petite 里的模板

在掌握了模板的基本用法之后，我们可以给 Petite 添加一个很酷的新功能：显示最近缩短的短链接列表。当用户访问 `http://localhost:8000/latest` 时，将看到最近 20 个被缩短的链接。为此需要创建一个模板，构建出最近链接的上下文，然后在页面上渲染出来。

首先，我们需要一个模板：

```
webmachine/petite/day2/petite/priv/latest.html.mustache
```

```
<!DOCTYPE html>
<html>
  <head>
    <link href="/css/bootstrap.min.css" rel="stylesheet" type="text/css">
    <link href="/css/petite.css" rel="stylesheet" type="text/css">

    <title>Latest Links</title>
  </head>
  <body>
    <div class="navbar navbar-inverse">
      <div class="navbar-inner">
        <a class="brand" href="/">Petite</a>
      </div>
    </div>

    <div class="container">
      <p>
        The latest shortened links are:
        <ul>
          {{#links}}
            <li>
              <a href="{{ short_link }}">{{ short_link }}</a>
              =>
              <a href="{{ long_link }}">{{ long_link }}</a>
            </li>
          {{/links}}
        </ul>
      </p>
    </div>
  </body>
</html>
```



```

|| {ShortLink, LongLink} <- LatestLinks],
Context = dict:from_list([links, LatestDicts])),

Response = mustache:render(TemplateStr, Context),
{Response, ReqData, State}.

```

资源和之前的列表例子非常相似。高亮的部分告诉资源从 `petite_url_srv:get_latest` 函数取得数据，而不是之前的硬编码列表。

5.3.4 处理多种内容类型

一个漂亮的页面固然很友好，但是有些服务需要同时支持人类和机器访问同一份数据。资源消费者对数据展示有着不同的喜好，在不同的情况下，API 需要展示不同的内容。例如，有些客户端希望得到 XML，而有些客户端则希望得到 JSON。

昨天我们看到如何为同一个资源提供多种展现，同时返回了“hello,world”的文本格式和 HTML 格式。今天我们将继续深入这个例子，研究一下如何提供最近链接列表的纯文本和 JSON 格式。

首先，我们给 `to_text` 和 `to_json` 添加一个 `content_types_provided` 函数：

```

webmachine/petite/day2/petite/src/petite_latest_resource.erl
-export([content_types_provided/2, to_text/2, to_json/2]).

content_types_provided(ReqData, State) ->
  [{"text/html", to_html},
   {"text/plain", to_text},
   {"application/json", to_json}], ReqData, State}.

```

我们先看看 `to_text`：

```

webmachine/petite/day2/petite/src/petite_latest_resource.erl
to_text(ReqData, State) ->
  {ok, LatestLinks} = petite_url_srv:get_latest(20),
  Result = lists:map(
    ① fun({Code, Link}) ->
        [base_url(ReqData), Code, " ", Link, "\n"]
      end,
    LatestLinks),
  {Result, ReqData, State}.
  ② base_url(ReqData) ->
    Host = wrq:get_req_header("host", ReqData),
    "http://" ++ Host ++ "/".

```

① 每一个短链接以及它的原链接都显示在一行里。

② `base_url` 助手函数使用 HTTP 头部的主机信息确定服务器的主机名和端口，所以不会被硬编码在源代码中。

这是 JSON 的版本：

```
webmachine/petite/day2/petite/src/petite_latest_resource.erl
```

```
to_json(ReqData, State) ->
    {ok, LatestLinks} = petite_url_srv:get_latest(20),
    LinkList = lists:map(
        fun({Code, Link}) ->
            ShortLink = base_url(ReqData) ++ Code,
            {struct, [{<<"short_link">>, list_to_binary(ShortLink)},
                {<<"long_link">>, list_to_binary(Link)}]}
        end,
        LatestLinks),
    Result = mochijson2:encode({struct, [{latest, LinkList}]}),
    {[Result, "\n"], ReqData, State}.
```

① 一个 JSON 的列表被显示成一个普通的 Erlang 列表，但是一个 JSON 对象是一个特别的元组，以 `struct` 开始，包含一个键值对的属性列表。

② `mochijson2` 是 MochiWeb 提供的 JSON 库，Webmachine 的 HTTP 服务器就基于它。

每一个展示类型都是内嵌的，所以它不知道其他展示类型的内容。让我们测试一下新的展现格式。

```
$ curl --header 'accept: text/plain' http://localhost:8000/latest
http://localhost:8000/2 http://erlang.org/
http://localhost:8000/1 https://github.com/basho/webmachine
http://localhost:8000/0 https://pragprog.com/
```

```
$ curl --header 'accept: application/json' http://localhost:8000/latest
{"latest":[{"short_link":"http://localhost:8000/2","long_link":"http://erlang.org/"},
{"short_link":"http://localhost:8000/1","long_link":"https://github.com/basho/webmachine"},
{"short_link":"http://localhost:8000/0","long_link":"https://pragprog.com/"}]}
```

如果未来需要一个新的展现类型，你无须修改当前的展现代码就可以轻而易举地做到。你只需要为 `content_type_provided` 添加新的内容类型以及对应的展现函数就可以。

5.3.5 我们在第 2 天学到的

今天学到了如何把 HTML 模板集成到 Webmachine 里, 以及如何简单地为一个资源提供多种展现形式。也许你想知道, 为什么 Webmachine 对于构造 API 来说如此强大。

明天我们将完成两个通常在其他框架中很困难的任务: 缓存和身份验证。Webmachine 简单好奇的特质会让这些任务也非常简单。

第 2 天的自学

查阅

- 调查一些其他为 Erlang 设计的模板库, `erlydtl` 是另外一个有着不一样风格的流行选择。
- 研究 `src/petite_static_resource.erl`, 虽然没有在我们的文章里进行讨论。在你还没有看文档之前查阅一下任何一个 Webmachine 函数。

实践

- 给 Petite 添加持久化 (提示: DETS 是一个磁盘版本的 ETS, 它们有很相似的 API)。
- 添加自定义的短编码支持。例如, 一个用户可能想把 `https://gragprog.com/` 变成 `http://localhost:8000/prag`。
- 创建另外一个前端资源, 让用户不需要通过 API 就可以缩短链接。

5.4 第 3 天: 照亮 HTTP 的阴暗面

Webmachine 把 HTTP 请求建模成状态机的方法很独特。目前你看到的处理和和其他的 Web 框架中的处理类似。让我们看一些 Webmachine 特有的东西。

首先, 我们来了解一下缓存。HTTP 协议有很多支持缓存的特性, 包括过期和版本管理。Webmachine 给资源函数暴露了这些功能, 这些和你之前看到的 HTTP 其他部分的处理方式一样。接下来, 我们将看到 HTTP 身份验证。虽然其他的一些框架也

支持身份验证，但大部分都不支持基于 HTTP 的身份验证，也不对外开放这样的功能。对于 Webmachine 来说，这只是 HTTP 协议的一部分。对状态机来说这是非常合适的，它为你的应用暴露了更多的控制。

5.4.1 让资源可缓存

现代 Web 应用都被寄望于在海量的设备和网络中实现闪电般的访问速度，和其他应用相比通常支持更多的同时在线用户。缓存资源是一种常见的提高应用访问速度和伸缩性的方法。

在大多数框架里，缓存都是事后才考虑的，在最好的情况下，也仅是一个高级功能。Webmachine 拥有内置的缓存支持，非常易于使用，所以可以利用它给你的用户创建一个优化和快速的访问体验。

HTTP 为浏览器提供了一些不同的缓存类型，我们将看到 `last modified` 报头、`ETag`、`expire` 报头和缓存控制指令。有了这些小的工具，你可以根据使用场景为每个资源添加不同的缓存行为。

Last Modified 报头

Last Modified 报头就像它描述的那样，告诉客户端资源最后修改的时间。例如，如果用户的资料从上周四开始都没有修改，Last Modified 报头就会把上周四作为最后修改的时间放到 HTTP 响应的报头里。Web 浏览器或其他 HTTP 客户端如果发现一个资源有了最后修改时间，就可以有条件地使用 If-Modified-Since 报头请求资源，客户端将只获取这个时间之后修改的资源。

所有的这些逻辑都已经在客户端实现了，所以现在要做的只是在被请求的时候为 Last-Modified 报头生成日期和时间。在 Webmachine 里，只要实现 `last_modified` 资源函数就可以完成这些事。Webmachine 和 HTTP 客户端会处理接下来的事情。

让我们创建一个叫作 `cache` 的 Webmachine 应用来实验缓存功能。接下来，修改 `cache_resource` 为它添加最后修改报头：

```
webmachine/cache/src/cache_resource.erl
-module(cache_resource).
-export([init/1,
        to_html/2,
```

```

        last_modified/2])).
-include_lib("webmachine/include/webmachine.hrl").
init([]) -> {ok, undefined}.

last_modified(ReqData, State) ->
    {{2013, 6, 12}, {22, 42, 00}}, ReqData, State}.
to_html(ReqData, State) ->
    {"<html><body>Hello, new world</body></html>\n", ReqData, State}.

```

`last_modified` 函数返回一个日期 June 12, 2013, 和时间 10:42 p.m. GMT。日期和时间必须是 GMT 时间。Webmachine 使用了三元组中的两个元组展示时间, 第一个元组用来展示日期, 第二个元组用来展示时间。

编译并启动服务器, 我们就会在命令行里看到正确的输出结果。

```

$ curl -i http://localhost:8000/
HTTP/1.1 200 OK
Server: MochiWeb/1.1 WebMachine/1.10.0
Last-Modified: Wed, 12 Jun 2013 22:42:00 GMT
Date: Thu, 13 Jun 2013 04:47:25 GMT
Content-Type: text/html
Content-Length: 43

<html><body>Hello, new world</body></html>

$ curl -i --header 'if-modified-since: Wed, 12 Jun 2013 22:42:00 GMT' \
> http://localhost:8000/
HTTP/1.1 304 Not Modified
Server: MochiWeb/1.1 WebMachine/1.10.0
Date: Thu, 13 Jun 2013 04:47:30 GMT

```

首先我们还是按一般的方式请求资源, 然后 Webmachine 返回这个资源并包含 Last-Modified 报头。下次当我们请求这个资源的时候, 就会提供 If-Modified-Since 报头做一个带条件的 GET 请求。如果资源没有被修改, 会返回一个空的 304 Not Modified 响应。

这个例子对这样小的资源来说有点浪费。但想象一下如果它曾经用来返回一个长会话的历史, 或者关于一个服务的详细配置信息, 取得资源是否修改的信息比取得信息本身代价要小得多。如果资源没有被修改, Webmachine 不会执行 `to_html` 函数或其他 `last_modified` 之后的函数。这意味着那些大开销的展现生成逻辑都不会被执行。

ETags

如果你已经记录最后修改时间或者可以计算最后修改时间, Last-Modified 报头就

已经足够好了。但是在很多情况下，它也无能为力。同样的 Last-modified 报头需要服务器时间同步，否则就失去了作用。为了避免这种时间带来的影响，Etags 强制使用字符串来表示一个资源的版本。

作为一个实际的例子，我们将为 `petite_latest_resource` 添加 Etags 的支持，它列出了最近用服务缩短的链接。如果使用 Last-Modified 报头，我们需要保存最后一次缩短链接的时间戳。这的确是可行的，但是 Petite 已经有合适的数据生成版本信息，那就是用作缩短链接的计数器。

让我们修改 Petite，为 HTTP 客户端返回一个合适的 Etag。当这一步完成，就像在最后修改时间的例子中那样，HTTP 客户端可以使用有条件的 IF-None-match 报头 GET 请求检查资源的更新情况。

首先，我们需要为 `petite_url_srv` 添加一个新的 API 调用，用来返回计数器和对应处理器的最后一个值。别忘了在导出函数列表的最上面添加一个新的 API 调用。

```
webmachine/petite/day3/petite/src/petite_url_srv.erl
get_last_id() ->
    gen_server:call(?SERVER, get_last_id).
```

```
webmachine/petite/day3/petite/src/petite_url_srv.erl
handle_call(get_last_id, _From, State=#st{next=N}) ->
    {reply, {ok, N - 1}, State};
```

注意这些信息由于直接存储在服务的内部状态里，所以不需要额外计算。

现在我们有一个取得计数器值的方法了。我们将使用它资源版本的 ETag。现在我们只需要在 Webmachine 里实现 `generate_etag` 资源函数，Webmachine 就能帮我们处理其余的事情。

```
webmachine/petite/day3/petite/src/petite_latest_resource.erl
-export([generate_etag/2]).
```

```
webmachine/petite/day3/petite/src/petite_latest_resource.erl
generate_etag(ReqData, State) ->
    {ok, N} = petite_url_srv:get_last_id(),
    {integer_to_list(N), ReqData, State}.
```

就像你从代码中看到的那样，它并没有比之前 `last_modified` 的例子更复杂。除了资源缓存的展现方式各异外，这两种方法基本上是相同的。

为 Petite 添加了 Etag 支持之后，让我们重新编译来看看响应是如何改变的：

```
$ curl -I http://localhost:8000/latest
HTTP/1.1 200 OK
Vary: Accept
Server: MochiWeb/1.1 WebMachine/1.9.2
ETag: "1"
Date: Thu, 13 Jun 2013 05:39:26 GMT
Content-Type: text/html
Content-Length: 925
$ curl -i --header 'if-none-match: "1"' http://localhost:8000/latest
HTTP/1.1 304 Not Modified
Vary: Accept
Server: MochiWeb/1.1 WebMachine/1.9.2
ETag: "1"
Date: Thu, 13 Jun 2013 05:44:40 GMT
```

首先，是正常的请求资源。Webmachine 插入了带版本信息的 ETag 新报头，这个报头由 `generate_etag` 返回。接下来，一个带有 If-None-Match 报头的新请求带着同样 Etag 值请求资源。由于没有改动，所以又返回了 304 Not Modified 响应。

让我们添加一个新的链接，然后使用同样的 ETag 再次请求最近缩短的链接：

```
$ curl -X POST http://localhost:8000/shorten \
> --data 'url=https%3A%2F%2Fgithub.com%2Ferlang%2Fotp'
http://localhost:8000/2

$ curl -I --header 'if-none-match: "1"' http://localhost:8000/latest
HTTP/1.1 200 OK
Vary: Accept
Server: MochiWeb/1.1 WebMachine/1.9.2
ETag: "2"
Date: Thu, 13 Jun 2013 05:46:46 GMT
Content-Type: text/html
Content-Length: 1132
```

由于客户端提供的 ETag 值和服务器端当前的 ETag 并不匹配，所以 Webmachine 返回了一个包含新 Etag 的响应。

Last-Modified 报头和 Etag 允许浏览器有条件地请求之前已经请求过的资源。另一个缓存模式是在一个时间段中无条件地进行缓存。这个目标可以被另外两个缓存方法实现：过期（Expires）报头和缓存控制（Cache-Control）报头。

Expires 报头

想想一个资源既频繁修改，也频繁被请求，但并不总是需要被实时更新。在那些

例子里，你可能想要告诉客户端它们可以使用数据一段时间，之后应该重新请求。Expires 头部使用日期和时间告诉返回的资源数据何时过期。

给之前创建的 cache 应用添加一个新的 expire_resource:

```
webmachine/cache/priv/dispatch.conf
[{"expire", expire_resource, []}.
```

```
webmachine/cache/src/expire_resource.erl
-module(expire_resource).
-export([init/1,
         to_html/2,
         expires/2]).

-include_lib("webmachine/include/webmachine.hrl").

init([]) -> {ok, undefined}.

expires(ReqData, State) ->
    {{{2013, 6, 15}, {05, 11, 00}}, ReqData, State}.

to_html(ReqData, State) ->
    {"<html><body>Hello, new world</body></html>\n", ReqData, State}.
```

在新资源里我们已经实现了 expires，它是另一个 Webmachine 的资源函数。就像 last_modified，它需要返回一个元组展示日期和时间。这个特别的日期和时间在例子里应该在不远的时期被更新，但记得要是 GMT 时间。

过期逻辑有点比较难测是因为它没有带条件的 GET 操作可以使用，它的值完全是信息化的。浏览器会一直跟踪下载资源过期时间，当被请求的时候，如果过期时间还未到，则会抓取缓存中的数据。如果你想测试 expire_resource，打开你的浏览器网络面板，在新的标签页上请求 <http://localhost:8000/expire>，应该会看到浏览器发出了一个普通的请求。如果你再一次请求同样的 URL，将看不到任何发生，因为它重用了缓存数据。注意重新刷新浏览器将请求一个忽略过期时间的新拷贝，所以重新键入 URL 在同一个标签页替代用点击回退按钮模拟访问同一个资源。

缓存控制

最后一个缓存机制，我们将看到的是 Cache-Control 报头。像 Last-Modified 报头一样，Expires 报头操作也有日期和时间，但是这些不总是容易提供的。Cache-Control 操作更像 ETag 那样可直接提供相关信息。例如，Webmachine 只需说明 5 分钟之后过

期，而无须说明具体的时间。

Webmachine 目前没有 `cache_control` 这样的资源函数可以用，但是我们可以在 `expires` 的函数添加缓存控制功能。替代返回一般日期和时间的元组返回的是 `undefined-Webmachine` 对 `expires` 的默认答案，并且 `wrq` 模块用于修改响应报头。

给 `cache` 应用例子添加新的 `control_resource` 函数：

```
webmachine/cache/priv/dispatch.conf
[{"control", control_resource, []}.
```

```
webmachine/cache/src/control_resource.erl
-module(control_resource).
-export([init/1,
         to_html/2,
         expires/2]).

-include_lib("webmachine/include/webmachine.hrl").

init([]) -> {ok, undefined}.

expires(ReqData, State) ->
    ReqData2 = wrq:set_resp_header("Cache-Control", "max-age=30", ReqData),
    {undefined, ReqData2, State}.

to_html(ReqData, State) ->
    {"<html><body>Hello, new world</body></html>\n", ReqData, State}.
```

现在 `expires` 函数设置了 `Cache-Control` 报头，同时制定了 `max-age` 属性为 30 秒。这让浏览器知道需要缓存资源最多 30 秒，之后要检查新的资源版本是否可用。你可以用于测试 `control_resource` 同样的方法测试 `expire_resource`，使用浏览器的网络查看器。它应该不会在 30 秒之内请求资源，即使你重复请求。

`Cache-Control` 头部非常易于使用，因为你经常知道多久展示层是有效的。它存储每一步都计算当前时间，添加进时间偏移，然后返回一个日期和时间给客户端。它有很多其他的属性，使你可以进行更复杂的缓存操作。

你当然可以在其他框架里添加缓存头部，但是它们只能让你手动操作 HTTP 头部。有些框架的中间件提供缓存支持，但经常是对应用进行整体配置，不能对资源进行配置。`Webmachine` 暴露了 HTTP 缓存的功能作为状态机模型的一个部分，这样就让它非常简单地为每一个资源定制缓存行为。

5.4.2 身份验证

身份验证是 HTTP 提供的功能，如果它不被完全暴露就很难使用。很多框架开始流行把身份验证相关的事情交给 Web 服务器来做。这让开发者的工作更轻松，但是向应用集成超出基本身份验证的功能更复杂了。虽然它只是另外一个 HTTP 功能，但 Webmachine 让它变得更简单。如图 5-4 所示，介绍了 Webmachine 状态机模型下的身份验证流程。

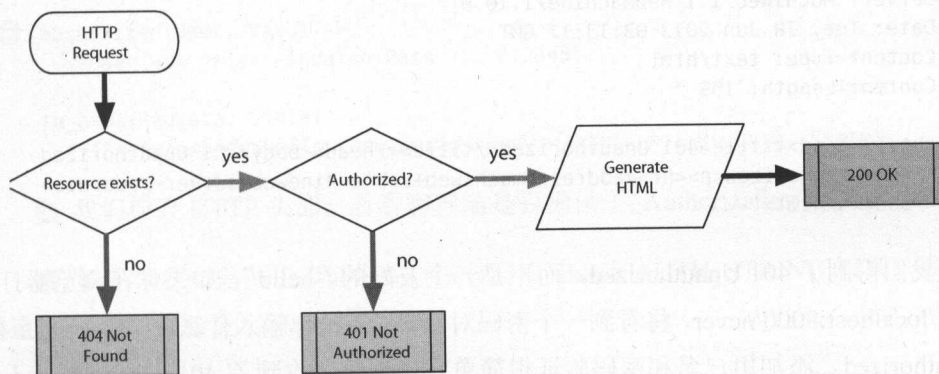


图 5-4 身份验证状态机

Webmachine 资源可以提供 `is_authorized` 函数用来检查当前请求是否已授权，如果是就返回 `true`，这样允许请求继续调用其他的资源函数。或者返回一个授权报头，会会在客户端返回的 401 Not Authorized 响应中添加一个 `WWW-Authenticate` 报头。

默认的 `is_authorized` 实现总是返回 `true`。如果你想观察一些更有意思的行为，可以创建一个总是返回一个授权头部的 `is_authorized` 函数。首先，创建一个叫作 `auth` 的 Webmachine 项目，添加一个 `auth_never_resource`：

```
webmachine/auth/priv/dispatch.conf
[["never"], auth_never_resource, []].
```

```
webmachine/auth/src/auth_never_resource.erl
-module(auth_never_resource).
-export([init/1,
         is_authorized/2,
         to_html/2]).
```

```
-include_lib("webmachine/include/webmachine.hrl").
init([]) -> {ok, undefined}.
```



```
is_authorized(ReqData, State) ->
    {"Basic realm=testing", ReqData, State}.

to_html(ReqData, State) ->
    {"<html><body>Hello, new world</body></html>\n", ReqData, State}.
```

这个资源里没有什么复杂的内容，但是用 curl 命令会揭示一个新的行为：

```
$ curl -i http://localhost:8000/never
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm=testing
Server: MochiWeb/1.1 WebMachine/1.10.0
Date: Tue, 18 Jun 2013 03:13:17 GMT
Content-Type: text/html
Content-Length: 159

<html><head><title>401 Unauthorized</title></head><body><h1>Unauthorized
</h1>Unauthorized<p><hr><address>mochiweb+webmachineweb server</address>
</body></html>
```

我们得到了 401 Unauthorized，而不是一个友好的“hello”。如果你在浏览器打开 `http://localhost:8000/never`，将看到一个密码对话框。无论你输入什么，它都会返回 401 Unauthorized。添加用户名和密码验证很简单。当浏览器收到了 401 Unauthorized 错误和包含授权方法及数据域的 WWW-Authenticate 报头后，它会提示用户输入用户名和密码并放入 Authorization 报头再次尝试请求资源。

客户端发送的 Authorization 值是一个授权方法。在这个例子里，Basi 和一个 base64 加密了一个授权字符串用作 HTTP 的基本身份验证，这个字符串只用了 base64 加密后的用户名、冒号和密码。

让我们通过创建 `auth_basic_resource` 来实现最基本的身份验证服务。

```
webmachine/auth/priv/dispatch.conf
[["basic"], auth_basic_resource, []].
```

```
webmachine/auth/src/auth_basic_resource.erl
-module(auth_basic_resource).
-export([init/1,
         is_authorized/2,
         to_html/2]).
-include_lib("webmachine/include/webmachine.hrl").
init([]) -> {ok, undefined}.
is_authorized(ReqData, State) ->
    AuthHead = "Basic realm=Identify yourself!",
    1 Result = case wrq:get_req_header("authorization", ReqData) of
```

```

2      "Basic " ++ EncodedAuthStr ->
3      AuthStr = base64:decode_to_string(EncodedAuthStr),
        [User, Pass] = string:tokens(AuthStr, ":"),
        case authorized(User, Pass) of
            true ->
                true;
            false ->
                AuthHead
        end;
    - ->
        AuthHead
    end,
    {Result, ReqData, State}.

4 authorized(User, Pass) ->
    User == "test" andalso Pass == "12345".

to_html(ReqData, State) ->
    {"<html><body>Hello, new world</body></html>\n", ReqData, State}.

```

❶ 我们检查 HTTP 头部，看看客户端是否提供了 Authorization 报头。

❷ 如果客户端提供了验证报头，应该含有 Basic AUTH_STRING 格式。我们必须通过 base64 对验证字符串进行解码。

❸ 我们调用 authorized 验证用户名和密码，看看是不是真的通过了验证。

❹ 我们通过硬编码的值测试授权函数，当然可以想象这是个在数据库里查找用户名的函数。

让我们看看这个资源怎么响应命令行的请求：

```

$ curl -v -u test:12345 http://localhost:8000/basic
<<omitted output>>
> GET /basic HTTP/1.1
➤ > Authorization: Basic dGVzdDoxMjM0NQ==
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24
.0 OpenSSL/0.9.8x zlib/1.2.5
> Host: localhost:8000
> Accept: /*/*
>
➤ < HTTP/1.1 200 OK
< Server: MochiWeb/1.1 WebMachine/1.10.0
< Date: Tue, 18 Jun 2013 03:25:10 GMT
< Content-Type: text/html
< Content-Length: 43
<

```

```
<html><body>Hello, new world</body></html>
* Connection #0 to host localhost left intact
* Closing connection #0
```

你可以在客户端的请求里看到 base64 编码的鉴权字符串，服务器很高兴地对这个古怪的字符串响应了 200 OK。

5.4.3 我们在第 3 天学到的

Webmachine 相较于其他框架暴露了更多的 HTTP 协议，这可以被看作是对缓存和身份验证的支持。这些在其他框架中遗失的部分却被 Webmachine 作为它状态机模型的一部分，实现它们就像实现“Hello, World”一样简单。

为了展示 Webmachine 如何轻而易举地处理缓存逻辑，我们用了四种不同的缓存策略实现了缓存：Last-Modified 报头、Etags、Expires 报头以及缓存控制指令。有了这四种工具，我们就可以轻易地用一个资源函数实现任何缓存逻辑。

Last-Modified 报头和 Etags 允许浏览器基于你的最后一次请求有条件地请求资源。即使它们表示资源的改变方式不同，Webmachine 也让它们一样简单易用。

Expires 报头和缓存指令用来通知浏览器数据的有效时间，并经常被用来控制数据请求的频率而无须顾及数据更新的频率。

最后我们看到了 HTTP 身份验证，它经常被委托给 Web 服务器处理。Webmachine 可以直接处理它，使它更容易和你的应用集成。

第 3 天的自学

查阅

- 在你的应用里使用其他的 Cache-Control 属性。

实践

- 如果你已经在应用里实现过缓存，比较它和 Webmachine 的难易。
- 除了 Etags 外，使用 Expires 或者 Cache-Control 头部修改 `petite_latest_resource`。
- 让 Petite 支持用户验证，只有验证通过的用户才可以缩短链接。

5.4.4 对 Justin Sheehy 的采访

Justin Sheehy 是 Basho 的 CTO，他在 Riak 这样的分布式数据库系统上工作。他同时作为 Akamai 的架构师在 MITRE 的智能社区做研究项目，以修炼他的分布式计算技能。Webmachine 只是他用到的很多开源项目中的一个。

我们：Webmachine 对 Web 应用的独特观点从何而来？

Justin：我们在 Basho 的一小部分人（不仅仅是我和 Andy Gross）一起实现了 Webmachine 的最初版本。我们对其他框架擅自以某种方式强制使用 HTTP 协议的某些功能感到很沮丧，想要一些东西允许我们思考资源在底层系统之间的传输。这不是那种帮你修改 Cookie 的普通 CRUD 应用。Alan Dean 画出了一个流程图，它不仅是灵感来源，也成为了 Webmachine 执行流程的前身。之后，Bryan Fink 通过一个神奇的可视化调试器添加了决策流程图。当它成为一个“击中”开发者的东西后，那可能是这个系统的转折点。我不得不说，这些开发者已经离不开 Webmachine 了。

我们：在 Webmachine 的帮助下构建应用时，最有意思的事情是什么？

Justin：我们给 Webmachine 设定的目标之一是使基础架构以良好的方式成为 Web 的一部分，所以当更底层的架构成为我的最爱之一时，没有什么可惊讶的。Caleb Tennis 使用 Webmachine 构建了一个 Web 接口，用于一个数据中心的物理管理。它暴露的资源可以让系统管理员检查或者调整冷却水泵的马达速度。我认为这是很有意思的工作，Webmachine 清晰的资源模型借助自身就可以开发出这样的东西。人们用 Webmachine 构建的很多东西我都觉得很有趣，虽然有些不那么可见。这是因为它经常用来编写中间件或者基础设施系统，而这对面向浏览器的应用则不是那么有用。

我们：开发者从 Webmachine 或者 Erlang 里能得到什么益处？

Justin：我认为其中的一个好处是 Webmachine 在某种程度上可以让你避免错误地使用 HTTP。即使有了正确性的标准，我们也无法让每个人都能正确地做每件事。但是很多 Web 框架都是围绕 HTTP 抽象而不是基于 HTTP 本身，这让用户无法告诉你相同的规范中构建的系统之间的互操作性。有了 Webmachine，你更希望去构建一个设计良好并且支持互操作的系统。而随着时间的推移，开发人员则希望利用更多 HTTP 有趣的功能和优势。Webmachine 能够很容易地做到这些，而不

需要“突破”框架。

Erlang 的优势可以单独作为一个采访，但这也在很大程度上取决于问题域。它说明我们看到了 Erlang 在 Web 开发上的优势，否则不会把 Webmachine 摆在首位。

我们：Webmachine 已经被其他不同的语言广泛效仿。其他社区从 Webmachine 中学到了什么？或者 Webmachine 从他们那里学到了什么？

Justin：这绝对是最高赞赏。当 Ruby、Clojure、Agda、Node.js 和其他语言的版本出现后，我们知道 Webmachine 已经造成了冲击。希望这些不同实现可以让 Webmachine 背后简单的核心想法帮助更多的开发人员。这样一个益处是对 Webmachine 自身来说，不同的社区有不同的期望，所以会提出很多不同的问题。我们绝对从 Webmachine 的社区里得到了很多更棒的点子，可以用来改进 Webmachine。

5.5 总结

Webmachine 做事的方式很特别，但是这种把 HTTP 作为状态机的独特设计让高级复杂的 HTTP 处理变得轻松简单。你无须猜测 HTTP 返回的状态代码或者创建复杂的视图，只要向 Webmachine 回答一些特定问题的答案，它就会帮你完成剩下的复杂决策。Webmachine 更像是你应用的“惊险岔路口”。

在这一章中，我们通过构建 Petite 这个缩短链接的小例子探索了很多 Webmachine 的功能特性。我们实现了返回不同格式类型的资源函数，管理重定向、缓存和身份验证。在 Webmachine 里，这些功能只需要写几行函数。但是在其他的框架里，这些功能都是一些高端特性，往往需要自己实现复杂的决策逻辑。

5.5.1 Webmachine 的强项

Webmachine 擅长作为后台系统。它经常用来把现有系统变成 HTTP 接口，无论是搜索引擎、数据库或者数据分析系统。Webmachine 让创建 HTTP API 变得更加轻松，使你能够专注在应用而不是繁冗的细节上。

由于 Webmachine 是通过 Erlang 编写的，这也让它在高可用性和大规模伸缩环境的问题上很有优势。这些正是大型后台系统一定会面对的问题。

Erlang 的轻量级进程模型给了 Webmachine 比其他框架更有助于并发执行的架构，这让它成为理想的服务间代理和服务集合，即使这些服务都采用不同的协议。

我们并没有在这章对这个特性进行说明，但是 Webmachine 内置的对流响应的支持让处理海量数据变得非常方便。

5.5.2 Webmachine 的弱项

虽然模板系统和其他关注于前端的功能都已经存在而且可以被集成到 Webmachine 里，但是它并不像典型的 CRUD 那样易用。它并没有太多的表单处理和数据库管理功能，但只有作为一个大型应用的时候人们才用 Webmachine 处理这些任务，应用后端才是 Webmachine 大展身手的地方。

对大多数人而言，Erlang 是一个神奇的语言，既有函数式的语义，也有非主流的语法。有些人对用函数式的思维去思考问题感觉很困难，例如使用循环的结构处理无状态的数据。

5.5.3 最后的思考

Webmachine 是 Web 应用中很独特的工具。状态机模型允许你用简单到不可思议的方法直接使用 HTTP 协议的全部功能。在这一点上，Webmachine 继承了 Erlang 的并发性、健壮性和伸缩性。这就像你在阅读《惊险岔路口》一样有趣。

第 6 章

Yesod

Yesod 可能是已有 Web 框架里最快的一个¹，但它的价值还远不止于此。Yesod 是基于 Haskell 语言的，Haskell 语言和其他动态语言一样极富表现力，其静态类型系统可以让 Java、C++ 看起来像个玩具一样。这个丰富的类型系统在 Yesod 里被反复利用，用来编写可以抵御程序员错误和恶意攻击的安全代码。

如果你之前听过 Haskell，可能会觉得 Haskell 写的 Web 框架就像一个万智牌游戏²，非常复杂并且充满了神秘的规则，甚至难以掌握。不要让这神秘感吓到你。其实 Yesod 更像 Set 纸牌游戏³，这是一个快节奏的纸牌游戏，虽然有一些重要的概念，但是学起来非常有趣，而且非常易懂。

Yesod 将其独特的思想带到 Web 开发中，基于强类型语言的思想而构建并对其进行了重组。如果你使用 Sinatra 或者 Clojure 的话，测试时可能会发现很多错误。但是如果使用 Yesod，它的编译器会在应用运行之前就帮你发现很多种错误。Yesod 利用 Haskell 的类型系统将一些领域概念编码成自己的类型，由编译器而不是程序员本身来保证程序的正确性。

¹ <http://www.yesodWeb.com/blog/2011/03/preliminary-warp-cross-language-benchmarks>

² http://en.wikipedia.org/wiki/Magic_the_gathering

³ http://en.wikipedia.org/wiki/Set_%28game%29

6.1 Yesod 简介

Yesod 是希伯来字里面“基础”的意思，这个框架致力于为你的应用提供一个强大而坚实的基础。本书中讲到的一些框架会关注如何探寻一个开发应用的新方法。而 Yesod 与它们不同，它使用了传统的架构并利用 Haskell 的优势来保证自己的稳定性。

6.1.1 组成部分

像其他流行的框架一样，Yesod 也有模型、视图和控制器。而它的独特之处，在于有一个强大的静态类型系统。查询模型的键值有不同的类型，这样你就不会将用户 ID “123” 和发票 “123” 混淆了。视图模板是已编译的，可以强制你去关注那些极容易被忽视的安全因素。控制器则处理那些不会过时的 URL。

Yesod 和其他框架一样有同样的组成部分，但是使用 Yesod 编程感觉完全不同。Haskell 是一个惰性函数式语言，这可能听起来非常陌生。而且，使用富类型系统来编写 Web 程序也不多见。从某种程度上说，Yesod 就是一个从奇怪而又熟悉的平行宇宙来的框架。

6.1.2 计划

接下来的 3 天里，你将会看到 Haskell 的类型系统是如何既像其他动态语言一样极富表现力，同时又可以避免一些常见错误的，从而使得你的应用足够健壮以抵御攻击。

第一天我们会学习路由和模型的一些基本知识，并且会开始构建我们这一章的应用——Rumbled 数据库的部分。

第二天我们会学习 Yesod 的模板语言——Hamlet 和 Lucius，以及它控件的抽象，从而为应用创建可复用的组件。然后我们会学习 Yesod 的声明形式。最后我们会学习认证和授权来结束这一天。

第三天主要是将我们所学到的东西集成到真正的应用中去，并利用前两天所学到的东西创建一个用户可以分享和评论的社交类新闻聚合网站。

这一周将会非常有趣并且充满挑战，让我们开始吧。

6.2 第1天：你不能搞错的数据

这一周我们将会创建一个新闻类聚和网站：Rumble。像 Rumble 这种专注于用户生成内容的网站一直以来都很容易被攻击，因为它们是将用户提交的不可信的内容结合起来生成动态的页面。开发者通过审核和限制用户输入来抵御这种攻击，但是动态语言基本上不能提供什么帮助，也不能确保每一种可能的情况都被覆盖。

今天我们会运行一个示例程序，并且开始学习 Yesod 是如何应用 Haskell 的类型系统来缓解这些问题的。最后我们会创建 Rumble 的模型，希望到时候你能体会到编译器为了保证代码的正确性和应用的安全性而做的工作。

6.2.1 新手入门

实践 Yesod 最简单的方法就是安装 Haskell 平台¹。Haskell 平台支持 Windows、Mac 和 Linux，而且它可能已经包含在你的包管理器里了。

Cabal 是大多数 Haskell 开发者都会用的构建工具，也包含在 Haskell 平台里了。只要安装 Haskell 平台，你就可以使用 Cabal 来更新它的包数据库和安装 Yesod：

```
$ cabal update  
$ cabal install yesod-platform yesod-bin persistent-sqlite
```

这样会分别安装 Yesod 框架、Yesod 命令行工具和 Yesod 数据库的 SQLite 后端。这个包里有很多依赖，Cabal 会帮你下载编译和安装。在我们这相对最快的机器上编译所有这些大概会花费 15 分钟，但在你的机器上可能会有所不同。

你可以使用 `yesod version` 命令来查看所有东西是否能正常运行，它应该会打印我们安装的 Yesod 的版本信息。本节使用 1.2 版本。

¹ <http://www.haskell.org/platform/>

```
$ yesod version
yesod-bin version: 1.2.1
```

装好 Yesod 以后，我们就可以开始创建第一个应用了。

6.2.2 Hello, World

yesod 命令行既可以用来创建一个新的应用，也可以在开发阶段用来编译和运行应用。

虽然 Yesod 应用可以手动创建，但是交给工具来做会更容易。你可以使用 `yesod init` 来创建一个新的应用。当出现提示时，输入 `hello` 作为应用的名字，输入 `s` 来选择 SQLite 数据库。

```
$ yesod init
Welcome to the Yesod scaffolder.
I'm going to be creating a skeleton Yesod project for you.
```

这样会创建一个新的目录 `hello`，并将你的新工程放在这下面。我们可以在应用的目录下运行 `yesod devel` 来编译和运行应用。

```
Project name: hello
Yesod uses Persistent for its (you guessed it) persistence layer.
This tool will build in either SQLite or PostgreSQL or MongoDB support for you.
We recommend starting with SQLite: it has no dependencies.
```

```
s      = sqlite
p      = postgresql
pf     = postgresql + Fay (experimental)
mongo  = mongodb
mysql  = MySQL
simple  = no database, no auth
url    = Let me specify URL containing a site (advanced)
```

```
So, what'll it be? s
That's it! I'm creating your files now...
<<omitted output>>
```

`yesod devel` 会监控应用的变化，并且在合适的时机重新编译应用。你在工作的時候可以让它保持运行状态，当需要退出的时候可以按回车键。

```
$ cd hello
$ yesod devel
Yesod devel server. Press ENTER to quit
<<omitted output>>
```

当第一次启动时,你会注意到它为应用创建了一些数据库表。当你使用浏览器访问程序时,它也会打印一些日志信息。我们可以访问一下 `http://localhost:3000/`, 去看看默认的 Yesod 结构。

现在让我们来给默认的应用添加一个“Hello, World”路由和处理器。`yesod` 命令行还有一个子命令 `add-handler`, 可以给应用添加新的路由和控制器模型。让我们来用它给 `/hello` 添加一个路由:

```
$ yesod add-handler
Name of route (without trailing R): Hello
Enter route pattern (ex: /entry/#EntryId): /hello
Enter space-separated list of methods (ex: GET POST): GET
```

这会在 `hello/config/routes` 里创建一个新的条目:

```
yesod/hello/config/routes
/static StaticR Static getStatic
/auth AuthR Auth getAuth
/favicon.ico FaviconR GET
/robots.txt RobotsR GET

/ HomeR GET POST
➤ /hello HelloR GET
```

最后一行就是由 `yesod add-handler` 添加的。通过文件可以猜测出来,每个路由单独列为一行,并由路径、控制器名称和一系列可被接受的方法组成。我们新建的路由指向一个名叫 `HelloR` 的新的控制器,你可以在 `hello/Handler/Hello.hs` 找到它:

```
yesod/hello/Handler/Hello.hs
module Handler.Hello where
import Import
① getHelloR :: Handler Html
② getHelloR = return "Hello, world!"
```

① 处理器的方法的名字一般是 `methodHandlerR` 这种模式,这里的 `method` 是 `GET` 或 `POST` 这样的 HTTP 方法,而 `Handler` 是这个处理器模块的名字。方法名称按惯例以 `R` 为后缀,来说明它是处理路由的方法。“`Handler Html`”的类型签名指出这个方法将会生成一个什么样的类型。在这里 `HTML` 就是生成出来的,如果改变基于请求信息类型的话,也可以生成其他类型。

② 这样将没有处理过的 `HTML` 以字符串返回,可能是最简单的一种方法。

如果 `yesod devel` 还在运行，它会检测到新的控制器并且自动重新编译和加载应用。如果你通过浏览器访问 `http://localhost:3000/hello`，就可以看到来自自己的亲切问候。

6.2.3 使用 DSL 描述数据

`config/routes` 文件是一个由纯文本写成的路由列表。它是由一个非常简单的 DSL 写成的，这个 DSL 是由 Yesod 转换成 Haskell 然后编译进你的应用里。Yesod 里面有很多 DSL，我们在这章会继续学到。接下来我们要学习的是 DSL 数据建模。

Yesod 的数据库叫作 Persistent。它同时支持 SQL 和 noSQL 数据库，以及它自己用来定义模型的 DSL。但是比起对象关系映射（object-relational mapper, ORM）的复杂性，它相当的灵活。而且正如我们所见，它完全是安全的。

我们的新闻类聚和网站——Rumble 只需要一个简单的数据模型：用户发布新闻，其他的所有用户可以在这些新闻后面进行评论。模型都以一种简单的格式存储在 `config/models` 里，下面让我们一起看一下 Rumble 的模型文件：

```
yesod/rumble/v1/rumble/config/models
```

```

1 User
2   ident Text
3   password Text Maybe
4   UniqueUser ident
5   deriving Typeable Show

Post
  title String
  url String
  author UserId
6  score Int default=0
  created UTCTime default=now
  deriving Typeable Show

Comment
7  post PostId
  author UserId
  created UTCTime default=now
  body Text
  deriving Typeable Show
```

- ① 模型是通过名字以及一些缩进显示的字段列表来定义的。这里是一个创建好的 User 模型，每个模型都定义了一个 Haskell 类型。Haskell 里的类型都是首

字母大写，以区别于其他的变量和方法。类型以及其他标识符都是以驼峰格式书写的。

- ② 一个字段由其小写的名字和类型来指定。Text 类型是一个简单的字符串。其他的通用类型有：Int、Bool 和 UTCTime。
- ③ 类型之后是选项。这里我们添加了 Maybe 关键字，将 Haskell 中 Maybe 类型的值封装了起来，这个值可能是 Just Text 或者 Nothing。这就是如何表现数据库表的空列的方式。
- ④ 这是一个唯一性约束，因为它是以大写字母开始的。这里给出了一些字段的列表，这些字段的组合必须对于这个模型是唯一的。在这里，我们要求 ids 必须是唯一的。
- ⑤ 我们使用 Haskell 的派生语法来让编译器自动生成 Typeable 和 Show 类型的实现类。Typeable 使得这个模型可以被存储在 Yesod 的缓存中，而 Show 让我们可以将模型打印出来，以有助于调试。而且，我们的所有模型都会用到这两个类型。
- ⑥ 默认的选项允许设置初始值，这主要是用在迁移的时候。
- ⑦ post 和 author 是一组外键关系。PostId 和 UserId 这两种类型是由 Post 和 User 模型生成的，而且分别代表这两个模型的主键。要注意，ID 和模型是绑在一起的，这样你就不会不小心使用 PostId 来查询用户，类型系统会阻止你犯类似的错误。

因为默认的样例结构中没有包含 UTCTime 类型，所以我们得告诉 Haskell 的包管理器 Cabal 我们需要 time 包。首先将这个包添加到 rumble.cabal 的包列表中。下面的这个列表显示了 rumble.cabal 的一小部分，这样你就可以看到新的依赖添加到了哪里：

```
yesod/rumble/v1/rumble/rumble.cabal
```

```
, warp                >= 1.3          && < 1.4
, data-default
, aeson
, conduit              >= 1.0
, monad-logger         >= 0.3
, fast-logger          >= 0.3
➤ , time               >= 1.4
```

下一步需要将 `Data.Time` 引入 `Model.hs` 中，这样我们就可以在模型中使用 `UTCTime` 类型了：

```
yesod/rumble/v1/rumble/Model.hs
module Model where

import Prelude
import Yesod
import Data.Text (Text)
import Database.Persist.Quasi
import Data.Typeable (Typeable)
➤ import Data.Time

share [mkPersist sqlOnlySettings, mkMigrate "migrateAll"]
    $(persistFileWith lowerCaseSettings "config/models")
```

添加了高亮的这一行后，我们的模型就完成了。

6.2.4 使用模型

在工程路径下，我们可以运行 `ghci` 然后开始使用模型。构建模型比较简单易懂，每一个模型都有一个同名的构造方法。

```
$ ghci Model
GHCi, version 7.6.3: http://www.haskell.org/ghc/ :? for help
```

```
① *Model> User "foo" Nothing
User {userIdent = "foo", userPassword = Nothing}
```

```
*Model> let bar = User "bar" (Just "12345")
```

```
② *Model> userPassword bar
Just "12345"
```

- ① `User` 的构造方法需要两个参数。由于 `Password` 字段是一个 `Maybe` 字段，所以它既可以填 `Nothing` 来代表 `NULL`，也可以当有值的时候填任意字符串。
- ② 每个模型都会为它的每一个字段自动定义一个访问函数。在这里，`userPassword` 就是用来获取密码字段的。所有这些方法都是以 `modelField` 这样的格式命名的，比如想要获取 `ident` 字段的值，你可以使用 `userIdent` 方法。

现在我们已经可以构造一个模型对象了，接下来就要将它们插入数据库中。我们

需要用到 `runSqlite` 方法来执行数据库操作，`insert` 方法可以插入一条新的数据。

```

❶ *Model> import Database.Persist.Sqlite
    <omitted output>

❷ *Model> id <- runSqlite "rumble.sqlite3" $ insert (User "foo" Nothing)
    *Model> id
❸ Key {unKey = PersistInt64 1}

    *Model> :type id
    id :: Key User

```

- ❶ `runSqlite` 是由 `Database.Persist.Sqlite` 提供的，需要我们手动引入。
- ❷ `runSqlite` 的最后一个参数是一组可以运行的操作。这里只有一个操作 `insert`，它使用指定的模型对象在数据库中新建了一行数据，并且返回它的主键。
- ❸ 一眼看去，返回的主键像是一个泛型。但是检查了它的类型之后，你会发现它是关联在 `User` 模型上的。

从数据库中获取数据也很容易。首先，我们来看看两个可以返回单个对象的方法：`get` 和 `getBy`。`get` 需要传入模型 `m` 作为键值，并且会返回一个 `Maybe` 类型的 `m`。`getBy` 需要传入一个符合唯一性约束的关联的值，然后利用这个值去查找对象。它也会返回一个 `Maybe` 值，但它是 `Entity m` 类型的。下面我们就会看到：

```

❶ *Model> let withDB a = runSqlite "rumble.sqlite3" a

❷ *Model> withDB $ get id
    Just (User {userId = "foo", userPassword = Nothing})

    *Model> e <- withDB $ getBy (UniqueUser "foo")
    *Model> :type e
❸ e :: Maybe (Entity User)

    *Model> e
❹ Just (Entity {entityKey = Key {unKey = PersistInt64 1},
                entityVal = User {userId = "foo",
                                userPassword = Nothing}})

    *Model> import Data.Maybe
❺ *Model> userIdent (entityVal (fromJust e))
    "foo"

    *Model> withDB $ getBy (UniqueUser "missing")
    Nothing

```

- ① 首先定义一个帮助方法来让我们可以少打一些字。
- ② id 变量是我们在前一个例子中创建的 user 对象。这个结果会被包装在 Just 类型里返回。因为当键值没有存在于数据库中的时候，查询操作可能会失败。
- ③ 你可能认为 getBy 会像 get 一样返回一个 Maybe User，但是它返回的是一个封装在 Maybe 里的 Entity User。因为我们是通过一个唯一约束条件来查询的，可能并不知道对象的键值；而且由于这个键值不是模型的组成部分，所以会返回一个模型与键值结合在一起的实体。
- ④ Entity User 有两个组成部分：键，就是 Key User；还有值，也就是 User 对象。这两个部分的访问方法是 entityKey 和 entityVal。
- ⑤ 用户的 ident 字段可以被从实体里单独拿出来。fromJust 是获得 Maybe 类型的访问方法。

构造模型并将它们插入数据库，只是我们这个任务的一个部分。为了建立这个社交新闻网站，我们还需要一个查询功能。针对这个需求，Persistent 提供了 selectList 方法。

selectList 接收两个列表参数、一组过滤器和一组选项，并且会返回一组符合规则的实体。为了试用这些功能，让我们来添加更多的用户并为每个用户都添加一些新闻：

```
*Model> fred <- withDB $ insert $ User "fred" Nothing
*Model> jack <- withDB $ insert $ User "jack" Nothing
*Model> now <- getCurrentTime
*Model> withDB $ insert $ Post "Yesod" "http://www.yesodweb.com/"
    fred 17 now
*Model> withDB $ insert $ Post "Haskell" "http://www.haskell.org/"
    jack 103 now
*Model> withDB $ insert $ Post "Yesod @ Hackage"
    "http://hackage.haskell.org/package/yesod" fred 11 now
*Model> withDB $ insert $ Post "Persistent @ Hackage"
    "http://hackage.haskell.org/package/persistent" jack 5 now
```

现在我们来查询所有由 jack 发布的新闻：

```
*Model> posts <- withDB $ selectList [PostAuthor ==. jack] []
*Model> length posts
2
*Model> map (postTitle . entityVal) posts
["Haskell", "Persistent @ Hackage"]
```


过滤器使用的运算符和之前的非常相似，只是多了一个“.”后缀。如果列表中有多个过滤器的话，就必须都匹配上：

```
*Model> posts <- withDB $ selectList [PostScore >= 15, PostAuthor ==. fred] []
*Model> map (postTitle . entityVal) posts
["Yesod"]
```

如果你想获得那些满足任意过滤条件的记录，就在每一个过滤条件之间加“||.”：

```
*Model> posts <- withDB $ selectList ([PostAuthor ==. fred] ||.
  [PostScore >. 100]) []
*Model> map (postTitle . entityVal) posts
["Yesod", "Haskell", "Yesod @ Hackage"]
```

我们可以使用选项列表来对结果进行排序，使用 Asc 升序或者使用 Desc 降序，并且指定为哪个字段排序：

```
*Model> posts <- withDB $ selectList [] [Desc PostScore]
*Model> map (postTitle . entityVal) posts
["Haskell", "Yesod", "Yesod @ Hackage", "Persistent @ Hackage"]
*Model> map (postScore . entityVal) posts
[103, 17, 11, 5]
```

现在已经完成了增删改查操作中的一半功能，下面我们来看看如何更新和删除数据。

6.2.5 改变和删除模型

使用 Persistent 来更新和删除数据的方式有很多种。下面的例子是一种最简单的方法：

- ①

```
*Model> withDB $ update jack [UserPassword ==. Just "asdf"]
*Model> withDB $ get jack
Just (User {userId = "jack", userPassword = Just "asdf"})
```
- ②

```
*Model> withDB $ updateWhere [PostAuthor ==. fred] [PostScore +=. 10]
*Model> posts <- withDB $ selectList [PostAuthor ==. fred] []
*Model> map (postScore . entityVal) posts
[27, 21]
```
- ③

```
*Model> let post = (entityKey . head) posts
*Model> withDB $ delete post
*Model> posts <- withDB $ selectList [PostAuthor ==. fred] []
*Model> map (postScore . entityVal) posts
```

[21]

```

4 *Model> withDB $ deleteWhere [PostAuthor ==. jack]
*Model> posts <- withDB $ selectList [] [Desc PostScore]
*Model> map (postTitle . entityVal) posts
["Yesod @ Hackage"]

5 *Model> withDB $ deleteBy (UniqueUser "jack")
*Model> withDB $ selectList [] [Asc UserID]
[Entity {entityKey = Key {unKey = PersistInt64 1},
        entityVal = User {userIdent = "fred",
                          userPassword = Nothing}}]

```

- ① `update` 接收一个键和一组变化的列表，然后根据这个内容改变键值。注意，像过滤器一样，这里也使用了操作符加点的语法。但这里的操作符都是赋值操作，我们给 Jack 设置了一个密码。
- ② 如果你需要一次性更新多条记录，`updateWhere` 可以接受一组过滤条件和一组变化内容。在这个例子中，我们将 Fred 发布的所有新闻都提高十分。
- ③ `delete` 方法只需要一个键就可以删除一个特定的模型对象。在删除了那些 Fred 发布的置顶新闻以后，就只剩下一个新闻了。
- ④ `deleteWhere` 接收一个过滤器的列表，然后将所有满足条件的模型对象删除。因此，所有 Jake 发布的新闻都被删除了。
- ⑤ 如果你想删除一个对象模型但是不知道它的键值，`deleteBy` 可以接受一个唯一的约束条件。现在，Jack 这个用户被删除了。

这些就是 Rumble 创建和操纵数据而使用的所有工具了。这里面大部分内容都和其他框架一样，但是 Yesod 还是有一些不同。

在 Yesod 中，我们从没有用纯数字作为键值或者使用字符串来匹配唯一的记录。键都是强类型的，所以你就不会在需要用键来查询一个用户的时候不小心查到了一个新闻。即使数据库使用的是基本类型并且不能区分数字，Yesod 还是可以为你确保操作的正确性。

6.2.6 我们在第 1 天学到的

像往常一样，我们今天也是以“Hello, World”开始，并且学习了如何使用 yesod

工具来新建一个工程并为其添加新的控制器。另外，我们还预览了一下如何向浏览器输出结果。

然后我们学习了模型以及 Yesod 的数据库——Persistent。我们为自己的新闻类聚和网站——Rumble 定义了一个简单的结构，并且试着创建、查询、修改和删除模型对象。

接下来，我们看到了 Yesod 是如何使用 Haskell 的强大静态类型系统来在运行之前就保证了代码的正确性的。这是 Yesod 最突出的优势之一，后面我们还会看到更多它的优势。

明天我们将会学习 Yesod 的 Shakespearian 模板系统以及它用来管理视图的组件模型。

第 1 天的自学

查阅

- 一本 Yesod 的书，特别是关于 Persistent 的一章。
- Yesod wiki 上“powered by Yesod”这篇文章。

实践

- 使用 ghci 来为数据库添加一些“评论”模型数据。
- 在 Post 模型中添加一个“描述”字段。
- 试着给 runSqlite 或者 withDB 添加一系列的操作。当一个操作失败时，其他操作会怎么样？
- 尝试想出一个方式来破坏这个类型系统，将键值从一个模型伪造或者转换到另一个模型。可以做到吗？需要做多少工作？

6.3 第 2 天：视图、表单和认证

Web 框架都非常依赖于 HTML 模板系统，以方便地添加动态的输出。但是大多

数模板系统只是对字符串拼接做了简单的封装，因此它们非常容易将缺陷和安全漏洞引入你的 Web 应用中。比如，你可能会将一个错误的链接放进视图里或者允许用户创建的不被信任的内容来控制浏览器。

Yesod 里包含几个模板系统，虽然它们和你以前看到的非常相似，但是可以帮你避开那些其他模板系统没有避开的缺陷。在 Yesod 里，URL 和插入的文档内容都是安全的，也就是说你不需要在路由变化时关心视图的刷新，也不需要担心跨站脚本攻击。这需要一个特别的语法来引用 URL，但是这个模板的语法和其他模板系统很相似，所以学习成本很小。

今天我们来学习 Hamlet 和 Lucius，这是 Yesod 里两个关于 HTML 和 CSS 的模板语言。还有另一个语言，叫作 Julius，是 JavaScript 的模板语言。因为它和另外两个语言非常相似，所以我们不会专门来学习。我们在为 Rumble 构建前端的时候，还会见识到 Yesod 的表单处理能力。

6.3.1 Ye Olde 的模板语言

Yesod 的第一个模板语言是 Hamlet，可以生成 HTML。从某种程度上来说，Ruby 的 Haml 启发了 Hamlet¹，比如对空格敏感和试图让自己尽可能地少占用空间。

为了看清 Hamlet 怎么工作，我们来创建一个 Yesod 应用。运行 `yesod init` 命令，然后像我们之前做过的那样，在提示符后面选择 `wellmet` 作为工程名称、SQLite 作为数据库。然后我们创建一个新的控制器：`GreetR`。

```
$ yesod init
<<omitted output>>
$ cd wellmet
$ yesod add-handler
Name of route (without trailing R): Greet
Enter route pattern (ex: /entry/#EntryId): /greet/#Text
Enter space-separated list of methods (ex: GET POST): GET
```

请注意路由模式中的 `#Text`。它将模式参数化了，并且会调用 `getGreetR` 方法。`getGreetR` 会使用 Text 格式的参数来操作路由。我们还需要将 Text 这种数据格式引入，

¹ <http://haml.info/>

这样控制器才可以使用。

这些操作都是在 `Foundation.hs` 里做的，这个文件还定义了应用的数据类型。

```
yesod/wellmet/Foundation.hs
import Model
import Text.Jasmine (minifyM)
import Text.Hamlet (hamletFile)
import System.Log.FastLogger (Logger)
➤ import Data.Text (Text)
```

基础

Yesod 在希伯来语里是“基础”的意思，所以我们可以肯定一个 Yesod 的工程里会实现一个名为 `Foundation.hs` 的文件。这个基础模型定义了应用里的数据类型和基本配置。

它定义了 `App` 数据类型，这个数据类型将设置值、数据库配置、静态资源和其他一些与应用相关的特定的东西绑定在一起。然后它会创建一个 `Yesod` 类型的类的实例——还有一些其他的東西，从而控制应用的一些高级行为。

现在我们可以使用 `Hamlet` 来实现 `getGreetR` 了：

```
yesod/wellmet/Handler/Greet.hs
module Handler.Greet where

import Import

getGreetR :: Text -> Handler Html
getGreetR name = defaultLayout [whamlet|<p>Well met, #{name}! #{after}|]
    where after = "Good day!" :: Text
```

像我们之前在路由和模型里看到的一样，Yesod 的模板语言也是在 `Template Haskell` 里实现的。我们使用 `quasi-quote`（类符号）表达式 `[name|template|]` 在常规代码里生成 `Template Haskell`。这里面 `name` 是 `quasi-quote` 方法，`template` 是模板代码。在这个例子中，我们使用 `whamlet` 将 `Hamlet` 模板转换成 `HTML`。

`Hamlet` 标签没有必要显式关闭，因为模板语言对空格是敏感的。在模板里，可以使用 `#{expression}` 来引用控制器作用域内的变量和方法。因为 Yesod 是静态类型，这就保证了你提供的表达式一定会返回可以被转化成 `HTML` 的东西。

如果你运行 `yesod devel` 并且访问 `http://localhost:3000/greet/developer`，就可以看到我们所期待看到的问候语。

试着访问一下 [http://localhost:3000/greet/%3Cscript%3Ealert\('uh oh'\)%3C/script%3E](http://localhost:3000/greet/%3Cscript%3Ealert('uh oh')%3C/script%3E), 这段 URL 试图将一段 JavaScript 代码 `alert('uh oh')`, 注入到页面上。Yesod 会忽略掉你想要转化成 HTML 的 Text 类型的值。虽然其他很多框架也可以做到这一点, 但是 Yesod 凭借着它的类型系统可以在编译阶段就做到。这么做最大的好处就是你不会忘掉这一点。

`defaultLayout` 方法将一个小组件作为输入, 并且将它嵌入网站默认的模板中去, 这个模板放在 `templates/default-layout-wrapper.hamlet` 里。小组件就是网页上一组表示任意内容数据的集合, 比如一个简单的小组件可能包含一个独立的组件的 HTML 和它的 CSS 以及 JavaScript。

这是一个强大的抽象单元, 代表着你不需要再记住一个指定页面都需要哪些资源, 所有内容都方便地组织在一起。让我们看看小组件是如何工作的。首先使用 `yesod add-handler` 来创建一个控制器, 叫作 `Greet2`, 同时还要创建一个路由 `/greet2/#Text`。然后修改 `getGreet2R`, 让它可以和下面的代码匹配:

```
yesod/wellmet/Handler/Greet2.hs
```

```
getGreet2R :: Text -> Handler Html
```

```
getGreet2R name = defaultLayout $ do
```

```
    color <- return ("blue" :: Text)
```

```
    setTitle "Greetings"
```

```
    toWidget [lucius]
```

```
        .greet { font-weight: bold; color: #{color}; }|]
```

```
    toWidget [whamlet|<p .greet>Well met, #{name}!|]
```

❶ `setTitle` 设置了页面的标题。

❷ `toWidget` 将内容添加到小组件中。首先我们添加一个可以产生 CSS 输出的 Lucius 模板。注意变量替换和 Hamlet 的工作方式是一样的。

❸ Hamlet 是用来产生 HTML 的。`.greet` 语法会展开成 `class="greet"`。如果我们使用 `#greet` 的话, 它会被展开成 `id="greet"`。

你可以访问 <http://localhost:3000/greet2/developer> 来查看结果。

通过编码来创建小组件可能会有点乏味, 所以 Yesod 还提供了 `$(widgetFile "foo")` 这样的语法。它会查找名字为 `foo.hamlet`、`foo.lucius` 和 `foo.lucius` 的模板, 然后将找到的所有模板结合成一个小组件。在开发者模式下, 当模板变化时它还会重新编译

一遍。下面的例子就是 Greet3 如何使用新的语法和它的模板：

```
yesod/wellmet/Handler/Greet3.hs
getGreet3R :: Text -> Handler Html
getGreet3R name = defaultLayout $(widgetFile "greet3")
    where color = "blue" :: Text
```

```
yesod/wellmet/templates/greet3.hamlet
<p .greet>
    Well met, #{name}!
```

```
yesod/wellmet/templates/greet3.lucius
.greet {
    font-weight: bold;
    color: #{color};
}
```

这段代码做的事情和 Greet2 控制器一样，但是它被很好地组织在几个不同的文件内。或许你已经可以想象出这在组织视图构建块方面有多大的优势。只要再将需要的小组件结合在页面中，Yesod 就可以自己判断出需要哪些样式文件和代码。

模板语言本身还有很多特性，我们今天晚些时候就可以看到。现在先来创建一些表单，以便我们可以获取一些数据并存在数据库里。

6.3.2 功能性表单

为了从用户那里收集数据，我们需要在视图中添加一个表单，并且在用户提交时处理这些表单数据。幸运的是，Yesod 包含了一组非常有用的工具，可以用来创建、验证和处理表单。

接下来我们会通过为 Rumble 创建一个非常简单的用户管理接口来学习一下 Yesod 表单的基本知识，将可以看到一组用户列表并且添加和删除用户。

首先把 Rumble 的工程地址改回来，并且添加一个 Users 控制器和/users 路由来响应 POST 和 GET 请求。运行 `yesod devel` 来启动服务器，并且要保证在我们写代码的过程中服务器是运行状态。

我们将在新的 Users 处理器中添加一些代码，来定义一个表单生成方法：

```
yesod/rumble/v2/rumble/Handler/Users.hs
```

```

1 userForm :: Form User
2 userForm = renderDivs $ User
3     <$> areq textField "ID" Nothing
4     <*> aopt textField "Password" Nothing

```

- ① 这个类型签名表示表单中封装了一个 User 模型对象。
- ② renderDivs 会为表单生成 HTML 内容。
- ③ 这个表单是一个可适应的 (applicative) 表单，即它是由 Haskell 的 Control.Applicative 模型创建的。可适应是指你可以按顺序操作 (添加表单字段)，但是不能像我们在数据库查询或小组件构建中那样做任何变量绑定。<\$> 和 <*> 是 Control.Applicative 定义的操作，得出的结果是一个漂亮的内部 DSL。areq 方法定义了一个必填的字段，它的参数定义了字段的类型、字段的标签以及默认值。
- ④ 你可能已经猜到 aopt 定义了一个选项字段，猜得没错。将 areq 和 aopt 结合在一起，你几乎可以创建出任何一种表单。

表单生成器自己并不会做很多工作，所以还需要给用户页面创建一个模板，表单会显示在用户列表之后。在 templates 目录下创建一个新的 Hamlet 模板 users.hamlet:

```

<h1>Users

1 $if null users
  <p>There are no users.
  $else
    <ul>
2     $forall Entity _ user <- users
      <li>#{userIdent user}
    <hr>

  <form method="post" enctype="#{enctype}">
3    ^{userFormWidget}
    <input type="submit" value="Add User">

```

- ① Hamlet 使用 \$if .. \$else ... 来做有条件的输出，这里我们判断了 users 变量是否为空。
- ② \$forall 就是 Hamlet 版的一个 for 循环。它轮询 users，查看每一个元素是否匹配 “<-” 左边的模式。这样就把用户绑定到一个实体值上，这个实体值可以用来打印用户 ID。

- ③ 正如`#{...}`可以将 Haskell 表达式渲染到 HTML 上一样, `^{...}`可以将表达式渲染到小组件上。在这里, `userFormWidget` 作为表单的一个部分被嵌入进来。

现在既然已经有了模板, 我们就可以在控制器的 `getUsersR` 方法里使用它了。首先需要在数据库中查出一个用户列表, 然后用它来处理我们刚才创建的模板:

```
yesod/rumble/v2/rumble/Handler/Users.hs
```

```
getUsersR :: Handler Html
```

```
getUsersR = do
```

- ① `users <- runDB $ selectList [] [Asc UserID]`
- ② `(userFormWidget, enctype) <- generateFormPost userForm`
- ③ `defaultLayout $(widgetFile "users")`

- ① Yesod 的控制器可以使用 `runDB` 来执行数据库查询操作。通过被编码在控制器里的类型, `runDB` 可以知道应该使用哪种数据库。我们昨天已经看到了查询语句, 它会返回所有的用户, 并根据用户 ID 按字母排序。
- ② 根据指定类型创建一个空的表单, 它会返回一个放在`<form>`标签里的表单生成的小组件和编码类型。接下来, 这些变量都会放进模板里去。
- ③ 剩下要做的事情就是使用 `defaultLayout` 来渲染小组件。

在浏览器中打开 `http://localhost:3000/users` 查看结果, 可以看到我们昨天在数据库里加的那些用户信息。如果你尝试提交订单, 就会看到错误信息, 因为 `postUsersR` 还没有实现。

控制器的最后一个部分就是处理表单的提交事件, 将用户信息添加到数据库然后重定位回到 Users 控制器中:

```
yesod/rumble/v2/rumble/Handler/Users.hs
```

```
postUsersR :: Handler Html
```

```
postUsersR = do
```

- ① `((result, _), _) <- runFormPost userForm`
- `case result of`
- ② `FormSuccess user -> do`
- ③ `_ <- runDB $ insert user`
- ④ `redirect UsersR`
- `_ -> defaultLayout [whamlet|whoops|]`

- ① `runFormPost` 接收并处理一个表单, 根据提交的数据填充表单字段, 最后会返

回结果——一个小组件和一个所使用的编码类型，不过我们只需要用到第一个返回值。在后台，`generateFormPost` 和 `runFormPost` 会插入和校验安全标识符来防止跨站点请求伪造攻击（CSRF）。

- ② 如果表单处理成功，就会包含我们可以使用的用户模型对象。
- ③ 用户对象被插入数据库中，`runDB` 会返回它的新的数据库 ID。
- ④ 我们将用户重定位到 `Users` 控制器上。注意我们没有直接使用 URL 而是使用了控制器的名称，这可以提供一个到路由的安全引用。如果 `Users` 控制器使用的路由发生改变，这个重定位仍然可以工作。更好的是，`Yesod` 需要我们在这里传一个路由的类型，如果我们传了一个原始的字符串，它就会编译报错。这种类型系统会预防那种由程序其他部分代码未来可能产生的变化而引发的潜在问题。

虽然这只是处理了一种最基本的表单，但还是为我们展现了 `Yesod` 很多独特的特性。我们只写了少量的代码，但是还是做了很多事情。我们以声明的形式创建了一个表单，使用 `Hamlet` 在小部件中嵌入小部件，预防了应用的 URL 变化而带来的路由的变化。最棒的是，我们不需要去测试应用是否有问题，编译器在代码运行之前就找出很多错误。

现在我们基本上把 `Rumble` 的所有组成部分都做好了，就剩认证和授权了。

6.3.3 认证和授权

认证就是检查用户的身份信息。无论用户是使用密码还是第三方服务来验证，目的都是确保他就是自己声称自己是的那个人。授权影响到用户可以做什么事情，他们是否可以创建和删除资源，他们是否可以看到特定的信息。很多框架把授权和认证混为一谈，但在 `Yesod` 里它们是被区别对待的。

区别对待这两者可以让事情变得更简单。控制器并不关心认证的机制，而只关心需要哪种类型的认证。大多数情况下如你所见，认证都是在控制器的外面以声明的形式完成的，这可以让控制器关注于它真正的任务。

这两个特性都写在应用的 `Foundation.hs` 文件里，这个文件会为应用设置好主要数

据框架。

认证

让我们一起看一下 Foundation.hs 里的配置变量：

```
yesod/authme/Foundation.hs
instance YesodAuth App where
  ① type AuthId App = UserId
  ② loginDest _ = HomeR
    logoutDest _ = HomeR

  ③ getAuthId creds = runDB $ do
      x <- getBy $ UniqueUser $ credsIdent creds
      case x of
        Just (Entity uid _) -> return $ Just uid
        Nothing -> do
          fmap Just $ insert $ User (credsIdent creds) Nothing
  ④ authPlugins _ = [authBrowserId def, authGoogleEmail]
    authHttpManager = httpManager
```

- ① 这个带着类型的 AuthId 告诉 Yesod 哪个模型 ID 会被用来代表一个被认证过的用户。
- ② loginDest 和 logoutDest 决定了登入登出以后重定向的位置，这里我们将两个重定向的目的地都设置成主页。
- ③ Yesod 使用 getAuthId 在一组给定的证书里查询 AuthId。当使用第三方的认证服务时，只需要在没有用户的时候为这些证书创建一个用户对象。
- ④ 默认的，Yesod 会打开 Persona 和 Google authentication 插件。

这种默认设定对很多情况都是极好的，而且比起其他框架那种设置一个内置的密码验证要更安全。因为密码保存或者传输很容易出错，而这种错误都会导致很严重的后果。

现在我们试着创建一个简单的样例网址来试试认证功能。使用 yesod init 来创建一个新的工程叫作 authme，选择 SQLite 作为数据库，修改 config/routes 文件，使其只包含一个给 HomeR 的 GET 路由，然后用以下代码替换 Home 控制器：

```
yesod/authme/Handler/Home.hs
{-# LANGUAGE TupleSections, OverloadedStrings #-}
module Handler.Home where
```

```
import Import
import Yesod.Auth

getHomeR :: Handler Html
getHomeR = do
  ① user <- maybeAuth
    defaultLayout $ do
      setTitle "AuthMe"
      $(widgetFile "homepage")
```

- ① `maybeAuth` 返回一个包含在 `Maybe` 里的被认证的实体，还存在一个 `maybeAuthId` 方法可以只返回 `AuthId`。如果你想强制用户必须都是被认证的或者将他们重定向到登录页面，就可以使用 `requireAuth` 和 `requireAuthId`。

我们还需要替换 `homepage` 模板。在你继续下一步操作之前，要确保已经删除或清空了 `templates/homepage.julius` 和 `homepage.lucius`。

```
① $maybe (Entity _ u) <- user
  <p>Logged in as #{userId u}
  <p>
  ② <a href=@{AuthR LogoutR}>Logout
  $nothing
  <a href=@{AuthR LoginR}>Login
```

- ① 我们尝试匹配用户。如果用户是被认证的，`maybeAuth` 会返回给我们一个 `Entity UserId User`。如果匹配上了，我们就会将用户的 `ident` 值打印出来。
- ② `Yesod` 提供了一个可以将 `LoginR` 或 `LogoutR` 作为参数接受的路由 `AuthR`，选择哪个参数取决于你希望用户是登入还是登出。

运行 `yesod devel` 之后，你可以访问 `http://localhost:3000/` 来试着用 `Persona` 或 `Google` 登录。当登录成功了，应该可以看到你的邮箱地址被打印出来，然后还可以登出。

我们可以放一个导向 `Persona` 登录系统的链接在页面右边来取代登录页面，这样可以更好看一些。另外的一个控制器和模板就是做这个事情的：

```
yesod/authme/Handler/Direct.hs
module Handler.Direct where

import Import
import Yesod.Auth
import Yesod.Auth.BrowserId
```



```

authLinkWidget :: Widget
authLinkWidget = do
  ❶ onclick <- createOnClick def AuthR
  ❷ loginIcon <- return $ PluginR "browserid" ["static", "sign-in.png"]
  ❸ [whamlet|<a href="javascript:#{onclick}()"><img src=@{AuthR loginIcon}></a>|]

getDirectR :: Handler Html
getDirectR = do
  user <- maybeAuth
  defaultLayout $ do
    setTitle "AuthMe Direct"
    $(widgetFile "direct")

```

- ❶ createOnClick 是 Yesod 的一个帮助方法。Auth.BrowserId 可以创建当你想要触发 Persona 登录的 JavaScript 代码（Browser ID 是 Persona 的一个旧名字）。
- ❷ PluginR 是一个访问插件资源 URL 的类型的 safely 方法。
- ❸ 我们的小组件变成了一个漂亮的图片以及一个可以触发我们创建的 JavaScript 的链接。

```

$maybe (Entity _ u) <- user
  <p>Logged in as #{userIdent u}
  <p>
    <a href=@{AuthR LogoutR}>Logout
$nothing
  ^{authLinkWidget}

```

这个和之前那个模板的不同之处在于我们在登录页面插入了 authLinkWid 小组件而不是一个简单的链接。

现在用户已经被认证了，我们还需要决定他们可以干什么。

授权

用于授权的主要控制点就是 Foundation.hs 中应用基础类型中的 isAuthorized 方法。

isAuthorized 接收一个路由和一个布尔值，这个布尔值代表请求是否是一个写请求，比如 PUT 或者 POST。它可以返回一个 Authorized，代表允许请求；一个 Unauthorized，代表拒绝请求；或者一个 Authentication-Required，会将用户重定向到登录页面。

这会使为路由设计一个授权变得非常简单。默认的 isAuthorized 实现会单纯地返

回一个 Authorized。

让我们快速地看着一个示例。在 `/secret` 使用 `yesod add-handler` 创建一个控制器 `Secret`，然后将高亮的这一行加入 `Foundation.hs`：

```
yesod/authme/Foundation.hs
```

```
authRoute _ = Just $ AuthR LoginR
```

```
> isAuthorized SecretR _ = do
>     maybeUserId <- maybeAuthId
>     return $ case maybeUserId of
>         Just _ -> Authorized
>         Nothing -> AuthenticationRequired
> isAuthorized _ _ = return Authorized
```

做完这些改变以后，试着在登录和登出的情况下访问 `http://localhost:3000/secret`。使用这些简单的工具，认证可以根据应用的要求编程简单或者复杂，而且不会妨碍到代码的其他部分。

6.3.4 我们在第2天学到的

我们今天学到了很多。首先，我们见识了 Yesod 的 Shakespearian 模板语言以及它的小组件抽象。不同于其他的模板系统，Yesod 的模板是类型安全的并且可以在应用运行之前进行编译。小组件将一个或多个模板结合起来形成了一个独立的、复用的单元，从而可以保证不同的组件有同样的风格和样式。

小组件的抽象不需要额外的开销，因为 Yesod 会将小组件编译进页面，好像它们一开始就被合并在一起一样。这样的好处在于所有的小组件都可以保持独立，而决定哪些小组件应该放在页面上的困难工作就被移交给了编译器。

接下来，我们看了 Yesod 的声明式表单并且使用它们来构建一个基本的新建用户页面。

最后我们学习了认证和授权，以及 Yesod 对几种知名认证系统的内部支持，还有如何将授权功能和控制器分离。

明天我们将把一切结合起来打造一个 Rumble 的可工作版本。

第2天的自学

查阅

- Yesod 的另一个 CSS 模板系统, Cassius 的一些例子。
- Hamlet 里一些选项属性的特殊语法。
- 可适应性表单、单一表单和输入表单的区别。

实践

- 把 wellmet 的控制器 Greet3 的名字变成一个由 fly 生成的唯一的类名称,而不是一个硬编码的类名称(提示:你可以在你的小组件里使用 `name<-lift newIdent` 创建一个唯一的名字)。
- 尝试使用 `isAuthorized` 的其他一些功能。试着重构一下最后的例子,让用户检查不需要在每一个子句中都重复一遍。

6.4 第3天:继续 Rumble

今天我们将把一切结合起来构建新闻类聚和网站。我们将使用 Shakespearian 模板来创建视图,并用 Yesod 声明式表单来处理新的内容。同时,我们还将看到如何修改默认的风格。我们将自上而下地先构建 Rumble 的新闻的头版信息,然后显示单个的新闻。这将大量使用到 Hamlet 的模板和小组件。然后我们会使用 Yesod 的声明式表单为新的新闻和评论添加表单。最后, Rumble 就可以做 beta 测试了,并且应该看起来如图 6-1 所示。

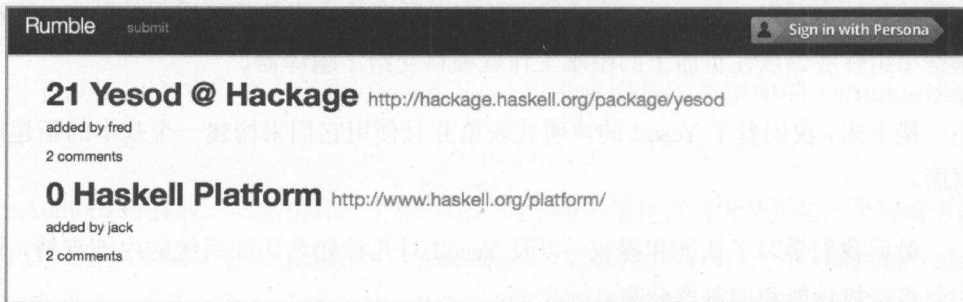


图 6-1 Rumble: 一个新闻类网站

6.4.1 创建头版

Yesod 的 `defaultLayout` 方法接收页面内容，一般情况下是一个小组件，然后将其放进网站的默认布局模板中进行渲染。生成的框架代码使用 `templates/default-layout-wrapper.hamlet` 作为默认模板。我们需要把它替换成更合适的内容，然后改变 `Home` 控制器来创建 Rumble 的新闻列表。

首先我们来创建所有需要的路由。创建好之后就可以使用它来在模板里生成类型安全的 URL 了，即使这个时候控制器还没有完全实现。使用 `yesod add-handler` 来创建以下路由和控制器。

创建一个 `Post` 控制器和一个 `/post/#PostId` 路由来响应 GET 请求。这会被用来显示单个新闻。

创建 `PostNew` 控制器和 `/new` 路由来响应 GET 和 POST 请求。这是用来显示新建新闻表单和处理新建新闻的。

创建一个 `Comments` 控制器和 `/post/#PostId/comments` 路由来响应 POST 请求。这是用来处理在某个新闻下新建评论的事件的。

如果一切都按计划进行了，你现在应该可以看到以下新的路由：

```
yesod/rumble/v2/rumble/config/routes
/post/#PostId PostR GET
/new PostNewR GET POST
/post/#PostId/comments CommentsR POST
```

```
① $doctype 5
<html>
  <head>
    <meta charset="UTF-8">

    <title>#{pageTitle pc}

    ^{pageHead pc}

  <body>
    <div .navbar .navbar-inverse>
      <div .navbar-inner>

        <div .container>
```



```

2      <a .brand href=@{HomeR}>Rumble

      <ul .nav>
        <li>
          <a href=@{PostNewR}>submit
        <ul .nav .pull-right>
          <li>
3            ^{pageBody authLinkContent}

      <div .container>
4        ^{pageBody pc}

```

❶ \$doctype 5 命令告诉 Hamlet 生成一个 HTML5 标签<!DOCTYPE>。

❷ 我们的导航栏包含了一个指向主页的链接和一个指向新建新闻表单的链接。注意我们使用了@{...}语法和路由名，这些是 Yesod 的类型安全的 URL。如果这些路由的 URL 变化了，这些链接仍然可以工作。而且当我们试图在@{...}构造中使用非路由类型的东西时，Yesod 编译器就会报错。

❸ 在这里我们引入了一个 Persona 登录按钮。如果我们不直接引用组件的话，默认的模板运行起来就会有一些不同。我们的 authLinkWidget 从昨天开始已经被转化成了类似于普通嵌入式内容的 PageContent。

❹ pc 变量是传给 defaultLayout 的页面内容。在这里主体组件被引入页面的内容区域中。

我们需要通过改变 defaultLayout 来设置 authLinkContent。将下面一行加入 defaultLayout，然后将 authLinkWidget 的定义加入 Foundation.hs 的 defaultLayout 的定义中去。

```
yesod/rumble/v2/rumble/Foundation.hs
```

```

defaultLayout widget = do
  master <- getYesod
  mmsg <- getMessage
  pc <- widgetToPageContent $ do
    $(combineStylesheets 'StaticR
      [ css_normalize_css
        , css_bootstrap_css
      ])
    $(widgetFile "default-layout")
  ➤ authLinkContent <- widgetToPageContent authLinkWidget
    giveUrlRenderer $(hamletFile "templates/default-layout-wrapper.hamlet")

```

如果你想查看到目前为止的结果，可以访问 <http://localhost:3000/users> 查看我们之前创建的用户管理页面。应该可以看到它漂亮的外观，头版应该包含了一列用户提交和评论的新闻。因为每一个新闻看起来都一样，我们可以使用小组件来处理它们的渲染。以下代码应该替换到样例代码中的 Home 处理器的部分：

```
yesod/rumble/v2/rumble/Handler/Home.hs
```

```
getHomeR :: Handler Html
```

```
① getHomeR = do
    posts <- runDB $ selectList [] [Desc PostScore]
    defaultLayout $ do
        setTitle "Rumble"
        $(widgetFile "home")
② generatePostWidget :: Entity Post -> Widget
generatePostWidget (Entity postId post) = do
③   (author, comments) <- handlerToWidget $ runDB $ do
        comments <- selectList [CommentPost ==. postId]
        [Asc CommentCreated]
④   author <- get404 $ postAuthor post
        return (author, comments)

$(widgetFile "post")
```

- ① 获取一系列新闻并把它们按照得分排序，再传给小组件用来显示。这太容易了！
- ② 为了渲染每一个新闻，我们还需要将一个 EntityPost 对象模型转化为一个小组件。除了 post 数据之外，我们还需要作者和评论信息。
- ③ runDB 方法通常在一个控制器内部运行。如果想要在一个小组件里使用它的话，可以用 handlerToWidget 来实现。
- ④ get404 方法可以方便地返回模型对象，还可以让处理器返回一个“404 找不到” HTTP 错误信息。

处理器本身没有包含太多信息，基本上只负责为模板设置环境。现在让我们看看模板：

```
yesod/rumble/v2/rumble/templates/home.hamlet
```

```
$if null posts
  <p>Nothing here yet.
$else
  $forall post <- posts
    ^{generatePostWidget post}

<article>
```

```

<header>
  <h1>
    <span .score>#{postScore post}
    ① <a href=@{PostR postId}>#{postTitle post} #
      <small>#{postUrl post}
    <p>added by #{userIdent author}
    <p>#{length comments} comments

```

- ① 请注意另一个类型安全的 URL 的用法@{...}。由于 PostR 路由需要接受一个参数，我们必须提供一个参数来生成安全的 URL。

除了变量解释以外，关于模板就没有更多的内容了。控制器设置好了环境，然后将渲染的工作代理给了模板。在这几行里我们就完成了很多工作。你可以访问 <http://localhost:3000/>，但是可能看不到太多的东西，因为还没有什么新闻发布。

现在，让我们一起创建新建新闻的表单吧！

6.4.2 创建一个发布新闻表单

我们为 Rumble 创建的表单应该比我们昨天创建的用户表单更复杂一些。生成表单的 `renderDivs` 方法，需要接收一个构造方法作为它的参数，并且当表单运行时，它会使用这个方法以及表单数据来构建一个对象。在用户表单里，我们使用 `User` 对象的构造器：Rumble 的表单需要的东西有些不同。

`Post` 对象的一个部分就是新闻的创建时间。这个不是由用户在表单里提供的，服务器应该在将新闻加入数据库的时候生成一个时间戳。这也就是说我们不能在调用 `renderDivs` 的时候使用 `Post` 的构造方法，而必须创建我们自己的构造方法。表单里会包含用户输入的数据，然后我们的构造方法会将这些数据和一些其他需要的数据结合起来生成一个真实的 `Post` 值。

下面展示的例子是新建新闻的控制器中 GET 的部分：

```

yesod/rumble/v2/rumble/Handler/PostNew.hs
module Handler.PostNew where
import Import
import Data.Time
import Yesod.Auth
① postForm :: Form (UserId -> Int -> UTCTime -> Post)
② postForm = renderDivs $ Post

```

```

        <$> areq textField "Title" Nothing
        <*> areq textField "URL" Nothing
getPostNewR :: Handler Html
③ getPostNewR = do
    _ <- requireAuthId
    (postFormWidget, enctype) <- generateFormPost postForm
    defaultLayout $(widgetFile "post-new")

```

- ① postForm 不会直接返回一个创建 Post 的表单，它返回的表单可以创建一个方法，这个方法接收一个 UserId、一个数值和一个 UTCTime 作为参数然后返回一个 Post。
- ② 我们在这里使用 Post 的构造方法，但是因为它只会接收五个参数的前两个，所以它只被部分应用到了。也就是说它最后会返回一个方法，而这个方法仍然需要后面的三个参数。
- ③ GET 控制器就很简单，会生成一个空白的表单然后渲染 post-new 小组件。注意对 requireAuthId 的调用，如果用户没有登录的话它就会把用户重定向了。

关于 post-new 模板没什么太多要说的，它只是把发布新闻表单的小组件封装了一下。

```

yesod/rumble/v2/rumble/templates/post-new.hamlet
<h1>New Post

<form method="post" enctype="#{enctype}">
  ^{postFormWidget}
  <input type="submit" value="Post">

```

新闻控制器的最后一步是在表单提交后对其进行处理，以下是 post Post NewR 部分：

```

yesod/rumble/v2/rumble/Handler/PostNew.hs
postPostNewR :: Handler Html
postPostNewR = do
    authorId <- requireAuthId
    ((result, _), _) <- runFormPost postForm
    case result of
①     FormSuccess makePost -> do
②         time <- liftIO getCurrentTime
③         post <- runDB $ insert $ makePost authorId 0 time
④         redirect $ PostR post
    _ -> defaultLayout [whamlet|whoops|]

```


- ① 要记住表单处理的结果是一个构造方法而不是一个 `Post` 值。
- ② 我们还需要 `Post` 的额外三个部分来创建它——时间、分数和作者。因为 `getCurrentTime` 是一个输入输出 (Input/Output, IO) 操作, 所以我们必须使用 `liftIO` 来调用它。
- ③ 根据所有获得的信息, 我们可以使用 `makePost` 来构建一个 `Post` 值然后把它插入数据库中去。
- ④ 现在任务完成了, 我们将用户重定向到发布新闻页面。我们再一次使用 `Yesod` 的安全 URL 来生成一个链接, 这样即使路由过一会发生变化, URL 也不会过期。

现在可以访问 `http://localhost:3000/new` 来给 Rumble 添加新的新闻, 不过接下来你会被重定向到一个没有被实现的页面上去。当添加了一个新闻之后, 你需要手动到主页面去查看结果。

我们今天的最终目标就是创建一个新闻处理器, 然后让这些新闻可以被评论。

6.4.3 查看新闻与提交评论

对新闻和提交评论控制器的实现都遵循和之前看到的主页及发布新闻控制器一样的模式。现在我们一起看看新闻控制器:

```
yesod/rumble/v2/rumble/Handler/Post.hs
module Handler.Post where
import Import
import Data.Maybe
import Yesod.Auth
import Handler.Comments (commentForm)
getPostR :: PostId -> Handler Html
getPostR postId = do
  authId <- maybeAuthId
  ① (post, author, comments, commentsWithAuthors) <- runDB $ do
    post <- get404 postId
    author <- get404 $ postAuthor post
    comments <- selectList [CommentPost ==. postId][Asc CommentCreated]
    ② authors <- mapM (get . commentAuthor . entityVal) comments
    return (post, author, comments,
    ③ zip (map entityVal comments) (map fromJust authors))
  (commentFormWidget, enctype) <- generateFormPost $ commentForm postId
```

```

defaultLayout $ do
  ④ setTitle $ toHtml $ (postTitle post) <> " - Rumble"
    $(widgetFile "post-full")
  generateCommentWidget :: Comment -> User -> Widget
  ⑤ generateCommentWidget comment author = $(widgetFile "comment")

```

- ① 控制器的主要功能就是从数据库中拿到所有必要的信息。我们必须手动获取相关数据，这样可能看起来很奇怪。这是因为 `Persistent` 不局限于关系型数据库，而且不会执行关系表示。这也就意味着一些常见的任务需要付出更多的工作。
- ② `mapM` 像 `map` 一样工作，只不过它还可以将元函数映射到列表上。这一点是必需的，正如数据库上的 `get` 操作一样。
- ③ 评论和作者被“压缩”在一起。`zip` 方法接收两个列表，并产生一个值组的列表，每一个值组都包含着每一个列表的一个元素。
- ④ 要注意我们不能直接把一个原始的字符串传给 `setTitle`，因为这样是不安全的。`Yesod` 要求它们可以被正确地转化为 HTML。
- ⑤ 就像我们之前看到的 `generatePostWidget` 一样，它也是一个帮助方法，可以将 `Comment` 的值显示在小组件上。因为 `Comment` 的值只包含了 `UserId` 的外键而没有了 `User` 值，我们还必须传一个作者对象。

`post-full` 模板和 `post` 模板非常相似，不过它还包含了所有的评论。如果用户登录了，一个新的评论表单也会显示出来。这和其他你见过的表单都一样，没有什么新的内容：

yesod/rumble/v2/rumble/templates/post-full.hamlet

```

<article>
  <header>
    <h1>
      <span .score>#{postScore post}
      <a href=@{PostR postId}>#{postTitle post}
      <small>#{postUrl post}
    <p>added by #{userIdent author}
    <p>#{length comments} comments

$maybe _ <- authId
<hr>
<form method="post" enctype="#{encType}" action="@{CommentsR postId}">
  ^{commentFormWidget}

```

```
<input type="submit" value="Post Comment">
<hr>
```

```
$forall (comment, author) <- commentsWithAuthors
  ^{generateCommentWidget comment author}
```

comment 小组件包含了两个模板，一个是给 HTML 的，另一个是给 CSS 的：

```
yesod/rumble/v2/rumble/templates/comment.hamlet
```

```
<article .comment>
  <header>
    <span .author>comment by #{userIdent author}
    <span .time>on #{show (commentCreated comment)}
    #{commentBody comment}
```

```
yesod/rumble/v2/rumble/templates/comment.lucius
```

```
.comment {
  background-color: #eee;
  padding: 10px;
  margin-bottom: 10px;
}
```

```
.comment header {
  color: #aaa;
  text-align: right;
}
```

最后，我们需要创建一个评论处理器和一个评论表单。评论处理器用来处理新闻页面提交的表单，然后将新闻的评论添加到数据库中去：

```
yesod/rumble/v2/rumble/Handler/Comments.hs
```

```
module Handler.Comments where
import Import
import Data.Time
import Yesod.Auth
```

- ① `makeComment :: PostId -> Textarea -> UserId -> UTCTime -> Comment`
`makeComment post body = \author time -> Comment post author time body`
- ② `commentForm :: PostId -> Form (UserId -> UTCTime -> Comment)`
`commentForm post = renderDivs $ makeComment post`
`<$> areq textareaField "Comment" Nothing`
- ③ `postCommentsR :: PostId -> Handler Html`
`postCommentsR post = do`
 `authorId <- requireAuthId`
`((result, _), _) <- runFormPost $ commentForm post`
`case result of`
 `FormSuccess mkComment -> do`
 `time <- liftIO getCurrentTime`

```

_ <- runDB $ insert $ mkComment authorId time
  redirect $ PostR post
_ -> defaultLayout [whamlet|whoops|]

```

- ❶ `makeComment` 是一个为评论创建构造方法的帮助类。不像 `Post` 那样前两个组件都是用户输入的，剩下的字段都是在处理的过程中填入的，`Comment` 的字段更分散一些。这就意味着我们无法使用 `Comment` 的一个部分作为构造方法。这个帮助方法只接收用户填写的字段，然后返回一个接收剩下字段的构造方法，最后返回一个构造好的 `Comment`。
- ❷ 有了构造方法，评论表单就变得很简单了。
- ❸ 最后，`POST` 控制器对应着一个新的新闻。我们将获取数据剩下的部分来完成 `Comment` 的值，然后调用返回的构造器。最终 `Comment` 的值被插入数据库中。

现在访问 <http://localhost:3000/> 就可以使用 Rumble 了。你可以在这里创建新的新闻和在新闻上进行评论。

6.4.4 我们在第3天学到的

今天使用我们所学到的所有知识来构建 Rumble 剩下的部分。我们将模板和小组件，表单和模型结合在一起构建 Rumble。最棒的是，多亏了 Yesod 对 Haskell 类型系统的使用，最后的结果可以避免很多对表单的攻击，还可以对未来的变化保持健壮。

开发者会非常开心，因为我们的应用速度快、占用空间小，既持有了编译语言的好处，又没有放弃动态语言的很多优势。用户也会非常开心，因为这个应用不会抛出奇怪的错误，因为很多错误在程序运行之前就被发现和解决了。

第3天的自学

查阅

- Github 上一些使用到 Yesod 的有趣的工程。
- 有关 Yesod 的博客或者 Yesod 的手册——它们都会有很多关于如何使用 Yesod 的例子。

实践

- 现在即使已经登录了，登录的链接还一直都显示在页面上。请你在用户已经登录时，将它转换成用户的电子邮箱地址显示出来。
- Rumble 没有给新闻投票的功能。新建一个新的处理器来处理投票的提交，并将它更新到新闻的得分上。改变模板，让它在主页和新闻页面包含一个投票的模板。
- 创建一个可以收集用户所有发布的新闻和评论的个人信息页面，将每个新闻和评论的用户名称链接到相关的个人信息页面上。

6.4.5 对 Michael Snoyman 的采访

Michael Snoyman 是 Yesod 的创造者，也是 FP Completed 的首席软件工程师。FP Completed 是一家决意推动 Haskell 成为主流的公司。他在个人开发上的追求使他转向 Haskell 并且创建了 Yesod。

我们：Haskell 杰出的类型系统让 Yesod 与众不同。在你开发的过程中，拥有静态类型系统会对你造成多大的影响？

Michael：为了充分利用强类型系统的优势，你必须多花一些时间提前设计一下你的类型系统。目的就是在类型中尽可能多地表示一些恒量，这样编译器就可以替你执行它们。根据使用弱类型系统（比如 Java）的经验，人们一般都会认为类型系统无法表示很多信息。而我使用 Haskell 的经验正与其大相径庭，这里有一些简单的例子。

- 将所有用户数据标记为不可信，可以抵御跨站脚本攻击。
- 基于表类型可以区分数值类型的数据库标示。
- 业务逻辑可以编码进变量里，比如“用户必须填写邮件地址或者电话号码，不能两个都空着”。

当初期的工作都做完以后，我们就可以看到很大的好处。你成功编码的任意一个恒量都可以完全依靠编译器来保证它的可靠性，这比单元测试更加强大。不

过测试还是可以保证在特定情况下属性值的正确性，软件的每一个部分里类型级别的需求都会被执行。

重构也变得非常简单。通常当我得知一个规格上的变化时，第一步就是改变数据类型。当这一步被正确地完成后，我就开始根据编译器的指示进行编码：首先我让编译器告诉我哪些需要被更新，然后我再去修改。很多情况下，当代码重新编译以后，应用还是可以正常地运行。

我们：Yesod 似乎为性能设置了一个很高的门槛，为什么性能对 Web 应用如此重要？

Michael: 这是一个很有趣的问题。当然了，Yesod 在基本的标准上表现得都很好，而且一些人也花费了大量的经历去优化 Yesod 基础架构的各个部分，从 Warp 网络服务器到我们的加密客户端会话 cookie（网络跟踪器）代码。这些都为提高用户体验做出了贡献。但是我真正的想法是，大多数时候一些合理的性能标准也是非常重要的。这是由多种因素决定的，比如用户的期望和程序本身的特性。一旦你超越了这个界限，比如开始考虑节省硬件消耗，这虽然好，但是并没有那么重要了。

我觉得更重要的是可扩展性，就是可以写出一个可以轻易扩展到多个内核甚至多个机器的程序的能力。这就意味着你可以应付用户量的迅速增长。Web 框架凭借它的无状态机制已经可以完成得很好，但是还不够。使用 Haskell 这样的语言会鼓励对不可变的数据结构和参考透通性的使用，大大简化了可扩展性实现的难度。有了强大的并发和并行的工具，比如软件事务内存和数据并行的 Haskell，也会对可扩展性有显著的帮助。

我们：你有没有发现 Yesod 创造了新的 Web 应用类型，或者让已有的类型更易于开发？

Michael: 就大部分而言，Yesod 并没有使用一些激进的新方法。Yesod 遵循了基本的模型-视图-控制器的网络框架设计，并且利用 Haskell 来提供可靠性、正确性和高性能。我的想法只曾在一方面非常具有革命性，那就是使用 Haskell。所以直接回答这个问题的话，我不认为我们创造了什么新的 Web 应用。至于使开发变得更容易，我发现比起其他流行的框架和语言，Yesod 在很大程度上减少了调试和维护大型程序的工作量。

一些相关的项目比 Yesod 更具革命性。功能反应式编程（Functional reactive programming, FRP）和基于持续的框架是 Haskell 社区正在开发的两个非常有趣的领域，它们倒是真的可能会创造一些网络应用的新形式。

我们：你看到过人们使用 Yesod 做的最有趣的事情是什么？

Michael：我看到过很多人用 Yesod 写出让人兴奋的 Web 程序，但是没有使用任何 Yesod 的后台功能（可能除了“服务器”端的 HTTP 响应头信息）。

有一个有趣的现象就是，Yesod 一直都被用来创建桌面应用的用户界面。由于 Yesod 可以将静态的资源 and Web 服务器嵌入到一个单一的可执行文件中，所以它可以创造这样一个程序——启动后台服务器进程，打开一个 Web 浏览器以查看用户界面。我们甚至可以提供一个库（wai-handler-launch）来让这种用法变得更简单，如此一来使得开发跨平台的图形化应用变得更容易了。这样类似的应用有：git-annex assistant（Git 附件助手）、一些 Wiki 平台、一个完整的个人财务管理系统和 Git 历史视图。

当然，还有很多使用 Yesod 写的“标准”网站。从个人博客到云上的一个成熟的 Haskell 集成开发环境（FP Haskell Center）。

我们：你对 Yesod 的将来有什么计划？

Michael：就目前来说，Yesod 是一个成熟的 Web 框架，而且满足了我最开始为它规划的那个问题领域后来又增加了几个。我对核心部分的计划就是持续提升产品质量和性能，并且继续为那些必要的安全隐患和问题做更新。但是我对当前的 API 很满意，所以我们的目标是尽量保证在向后兼容的变化中保证一个比较长的发布周期。我也从社区以及商业用户那里得到了很多反馈，我相信稳定性是我们必须向用户保证的。

即便如此，我们还是正在向周边一些新的领域扩张。Haskell 到 Javascript 的编译器、更好的客户端编码抽象、与更多更新的网络技术的集成（比如 WebSockets）和更强大的服务器段的特性的抽象（比如自动扩展和集群）等，这些都是在计划中的或者正在进行的。Yesod 的发展策略是鼓励在实验性的特性上进行快速迭代，在所有设计都集中起来以后，还鼓励与 Yesod 的核心技术进行深度的整合。我觉得这个进程可以继续下去。

当然，我们还积极地鼓励更广泛地从外界吸收。比如更全面的文档，不管是 Yesod 的书籍还是各种在线指引、帮助教程或者截屏都是对新用户非常有用的启蒙教材。在 FP Compute，我们还在持续改进 Yesod 的一些问题和 Haskell 语言的一些问题。我深信 Haskell 和 Yesod 有这样的潜力去帮助程序员写出更好更健壮的软件，而且希望大家都可以受益于这样的好处。

6.5 总结

Yesod 是一个由非传统语言构建的传统框架。Yesod 有模型、视图和控制器，但是同时还具有可编译、惰性计算和静态类型的特性。Haskell 对传统的网络框架做出了很多改进——安全性、速度和面对程序错误的健壮性。

很多你在测试的时候会发生的错误都会被编译器提前检查出来。在 Yesod 里，字符串和 HTML 是两种不同的类型，两者不能互相取代，所以你也不会在使用 HTML 的时候忘记给内容闭合。你也不会碰到空指针异常和未定义的结果，因为编译器会强制要求你处理 Maybe 值的所有结果。

在这一章，我们简单学习了一下 Yesod 的特性，并且创建了一个一般来说很容易出错的应用。因为有了 Yesod，编译器会复核代码，检查它是否按照需求运作，并且不会出现错误，所以我们自己就可以轻松多了。

6.5.1 Yesod 的强项

Yesod 充分利用了 Haskell 的类型系统来防止一些程序的错误。这对 Web 应用来说是很重要的，不但可以防止对用户满意度的损害，还可以预防一些会损害企业或用户的恶意行为。虽然这不是一个可以让你完全不需要测试的银弹，但是会帮你预防很多以前必须通过测试才能发现的问题。

而且 Yesod 速度非常快。在众多的 Web 框架里，它在速度之战中都是远远领先于其他框架的。更棒的是，在使用 Yesod 时，你不需要改变自己在使用动态语言的时候养成的习惯。

Yesod 的大多数库都非常丰富。模板语言在不牺牲类型安全的情况下，覆盖了绝

大多数的功能；身份验证插件支持 Persona、Google login 等多种验证方式；而且 Yesod 对 SQL 和 noSQL 数据库的使用也非常简便。

6.5.2 Yesod 的弱项

不可否认的是，Haskell 对于大多数程序员来说都十分陌生。它的惰性、纯函数和强类型的特性综合起来，还是需要一定的学习曲线的。

比起其他大多数框架，Yesod 的持久层库还是有点落后的。尤其是对连接 (join) 的缺失使得很多事情都变得非常复杂，尽管这样可以让处理 noSQLite 数据库变得更方便。

6.5.3 最后的思考

类型系统是一件美好的事情，如果你曾经受到那些能力不足的类型系统的困扰，那么 Yesod 的类型系统会让你得到弥补。Yesod 在任何一个可能的地方都用到了 Haskell 的类型系统，这样做并不是自作聪明。这个特性是其他语言实现的框架所不具备的，而且一旦你习惯使用后就很难摒弃了。

使用 Yesod 编程既轻松自由又让人沮丧。它把你从对安全性问题以及一些遗留问题的担心中解放出来。但是由于它会在你运行应用之前就找出大部分的问题，所以调试的痛苦就会集中在一段时间里。然而，今天你开发的时候经历的痛苦换来的却是以后用户脸上的笑容。

第 7 章

Immutable

Dwarf Fortress 是一款以深度和复杂度闻名的电脑游戏，同样出名的还有它不友善的界面和陡峭的学习曲线¹。数小时的教学视频只是为了帮你入门，但大部分人还是没开始多久就放弃了。坚持下来的人会享受到游戏丰富的玩法，但还是会对糟糕的界面感到失望。

再对比着来看 Minecraft²，一款受到 Dwarf Fortress 巨大影响的游戏。它有简洁的用户界面和有趣的图形效果，还保留着很多 Dwarf Fortress 中人们喜爱的有创意的游戏元素。

企业级 Java 网络开发有着丰富的功能，但被封装打包后不仅复杂难以使用，也难以学习。Immutable 会将这些相同的元素通过 Clojure 组合在一起，让事情变得简单，易于使用，便于入门。就像 Minecraft 将 Dwarf Fortress 中有创意的游戏元素带向大众一样，Immutable 的目标是将复杂的企业级开发工具变得易于理解使用。

7.1 Immutable 简介

网络框架通常构建在底层栈之上，负责处理 HTTP 交互。这个底层栈通常不含有

¹ <http://www.bay12games.com/dwarves/>

² <http://minecraft.net/>

其他的功能。这个设计推动了很多框架使用简单的架构，所有的代码都发生在网络请求的场景下。Immutant 扩展了这个底层栈，提供了更多的基元，如消息队列、守护进程、分布式缓存和计划任务。

7.1.1 Immutant 的特性

Immutant 基于 JBoss AS7，这是一个历史悠久的 Java 企业级版本应用服务器。在这样强大的基础上，Immutant 又加入了更多复杂的框架和应用。Rails 应用必须依赖于如 memcached 和 RabbitMQ 这样的外部服务，而 Immutant 不只提供了相同的服务，还将它们充分整合起来。这让它们十分易于使用和部署。

Immutant 还可以被集群在很多机器中，并且它的组件都可以正常工作。在缺省情况下，计划任务会只在集群中运行一次；缓存数据会被复制并分片。所有这些特性对小型创业公司的重要性和对大型企业是一样的。但在 Immutant 里，你不必应对 Java 的形式或在 XML 配置文件里花费时间。

因为 Immutant 应用是用 Clojure 写的，并且 Immutant 网络组件是基于 Ring 的（在第 4 章 Ring 中我们介绍过），所以我们会专注于其他的部分：消息队列、任务和缓存。我们会构建一个应用来监控一组 URL，并测试这些网页上的链接是否合法。这会需要异步处理和计划任务，并且需要使用缓存来保持高效率。

7.1.2 计划

第 1 天，我们会学习如何安装 Immutant 应用容器和将我们自己的应用部署到其之上。下一步，我们会学习怎样使用分布式缓存和怎样计划任务。同时，我们会使用缓存来存储获取到的网络页面结果。

第 2 天，我们会引入消息队列和管道。消息队列对于完成网络请求范围外的异步工作十分重要。管道是消息队列的一个抽象，它允许工作被分解成不同的步骤，每个步骤能并行运行并且以队列形式连接在一起。

最后一天，我们会看一下叠加，它可以让你在 Immutant 中混合多种语言以达到多语言编程的结果。你会看到 Ruby 和 Clojure 应用可以被部署在一起并且无缝交互。

我们还会了解一下集群，它会将你的应用分散在很多结点并更优雅地处理更多的数据和错误。

Immutant 提供了很多功能，让我们马上开始吧。

7.2 第 1 天：不仅仅是网络基础

现代 Web 应用正在不断地定义着软件的能力范畴，特别是当它达到一定规模后。去应付持续增长的复杂度和需求，开发人员不能只依靠数据库，还有内存支持、分布式缓存、消息队列和异步处理。但是，为去年的应用设计的全栈框架满足不了今日的需求，而 Immutant 弥补了这一不足。

今天我们要了解 Immutant 的基础，开始构建一个需要这些现代结构特征的链接监控应用 Overwatch。搭建好 Immutant 并运行起来之后，我们会构建获取页面和缓存结果的部分。

7.2.1 开始

使用 Immutant，你需要一个 Java 虚拟机 (Java Virtual Machine, JVM) 和 Leiningen 构建工具。你可以在第 4 章 Ring 第 1 天的开始获得更多相关信息。

有了 Leiningen 后，你需要在 Leiningen 用户文件中添加 lein-immutant 插件。将这个插件加到 `~/.lein/profiles.clj` 的列表中，如果没有这个文件和路径就自己创建一个，然后填入如下配置：

```
immutant/profiles.clj
{:user {:plugins [[lein-immutant "1.0.1"]]]}}
```

我们这里使用的 1.0.1 版本是编写这本书时的最新版本，但是你可以用更新的可用版本（请参考 Immutant 安装页面获得更多信息¹）。

你可以通过运行 `lein immutant` 来测试一切工作是否正常，输出的结果应该类似于如下所示：

¹ <http://immutant.org/install/>


```
$ lein immutant
```

Manage the deployment lifecycle of an Immutant application.

Subtasks available:

```
undeploy  Undeploys a project from the current Immutant
archive   Creates an Immutant archive from a project
deploy    Deploys a project to the current Immutant
run       Starts up the current Immutant, displaying its console output
env       Displays paths to the Immutant that the plugin is currently using
overlay   Overlays a feature set onto the current Immutant
test      Runs a project's tests inside the current Immutant

version   Prints version info for the current Immutant
install   Downloads and installs an Immutant version
list      Lists deployments or Immutant installs
```

Run ``lein help immutant $SUBTASK`` for subtask details.

Arguments: ([subtask] [project-or-nil subtask & args])

最后一步是安装 Immutant 应用服务器并启动:

```
$ lein immutant install
```

```
<<omitted output>>
```

```
$ lein immutant run
```

```
Starting Immutant: /Users/jack/.immutant/current/jboss/bin/standalone.sh
```

```
=====
<<omitted output>>
```

第一个命令下载并安装了 Immutant 发布的最新稳定版本。第二个启动了应用服务器。你应该会看到, 当服务器启动时输出了很多日志。如果你想停止服务器, 就按下[Ctrl-C]组合键。现在, 我们已经准备好创建和部署我们的第一个 Immutant 应用了。

7.2.2 Hello, World

我们可以像在 Ring 的章节 (Hello, World) 一样, 通过 `lein new` 创建一个简单的脚手架应用。我们还会创建一个额外的文件 `immutant/init.clj`, 这也是 Immutant 应用所需要的。让我们开始创建 hello 吧:

```
$ lein new hello
```

基于 ‘default’ 模板生成名为 hello 的项目。

lein new 创建的这两个重要文件是 Leiningen 项目的描述文件 `project.clj` 和我们应用的核心命名空间 `src/hello/core.clj`。首先，让我们在 `project.clj` 中添加 Compojure 为依赖：

```
immutant/hello/project.clj
(defproject hello "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.5.1"]
                 [compojure "1.1.5"]])
```

然后，我们会在 `src/hello/core.clj` 中使用 Ring 章节的“你好，世界”应用的相同代码：

```
immutant/hello/src/hello/core.clj
(ns hello.core
  (:use compojure.core))
(defroutes app
  (GET "/" []
    "Hello, World!"))
```

最后，我们需要使用 Compojure 的 `defroutes` 创建的 Ring 处理器来通知 Immutant 启动一个网络服务器。创建 `src/immutant/init.clj`，这是我们 Immutant 应用的初始化代码文件，添加如下代码至其中：

```
immutant/hello/src/immutant/init.clj
(ns immigrant.init
  (:require [immutant.web :as web]
            hello.core))
```

```
➤ (web/start "/" hello.core/app)
```

箭头标记行展示了怎样在 Immutant 应用中启动一个网络服务。我们提供根路径和 Ring 处理器，然后 Immutant 会负责其他的事情。在 Ring 的章节，我们通过在 `project.clj` 中添加特殊的元数据完成了相同的任务，其中 `lein-ring` 插件用来寻找主要的 Ring 处理器。Immutant 只是以不同的方式完成了相同的事情。

我们一会儿会看到，`Immutant.init` 不仅可以启动网络服务，还有其他的 service 如缓存、计划任务、消息机制等，这使得 Immutant 应用十分强大。

想在运行的 Immutant 服务器上部署应用，你可以使用 `lein immutant deploy`。部署完成后，你会看到一个简短的消息；几秒钟后，Immutant 应用服务器就会启动 `hello` 应用了。你可以在 `http://localhost:8080/hello/` 看到你的劳动成果。

7.2.3 分布式缓存

获取数据的代价很昂贵，特别是当数据存储于旋转的金属中时。即使是固态硬盘（SSD），也不会像 RAM 那样快。拥有众多用户的网络应用很快就不再满足于数据库存储，而必须在内存中缓存大量的数据。因为数据集庞大并且在持续增长，而且一台机器内存的容量相对来说是很小的，所以像 Memcached 这样的服务会将缓存分布在多台机器的内存中¹。

Immutant 在 `immutant.cache` 的命名空间下包含一个分布式内存缓存系统。它可以自动传播到整个 Immutant 集群中，甚至支持事务。在这部分，我们会了解一下怎样在缓存中存储和获取数据。我们还会了解一下记忆化，这是一种用来缓存函数结果的技术，在函数式编程中经常使用。

Immutant 是建立在 Clojure 之上的，因此可以享用供其他 Clojure 应用使用的很多交互开发功能。让我们创建一个新的应用来探索一下缓存中我们可以连接的 REPL。首先，创建一个新的应用叫 `overwatch`：

```
$ lein new overwatch
```

基于 'default' 模板生成名为 `overwatch` 的项目。

接下来，我们需要启动 REPL。可以通过添加 `:nrepl-port` 属性到 `project.clj` 的 `:immutant` 选项上来完成。之后我们还会需要一些额外的依赖：

```
immutant/overwatch/project.clj
(defproject overwatch "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.5.1"]
                 [org.clojure/tools.logging "0.2.6"]
                 [clj-http "0.7.6"]
                 [enlive "1.1.4"]]
  ➤ :immutant {:nrepl-port 0})
```

端口值为 0 表示 Immutant 应该选择一个随机端口号。继续运行 `lein Immutant deploy` 将应用部署到 Immutant 服务器上。当应用部署好后，Immutant 会将它选好的端口号写入名为 `.nrepl-port` 的文件，每个部署项目的 `project.clj` 文件同文件夹下都会生

¹ <http://memcached.org/>

成这个文件。

我们要做的其他事情是：添加依赖关系、开启服务或重新加载项目，可以在 REPL 里完成它们。你可以使用 `lein repl` 命令来开启一个 REPL 客户端：

```
$ lein repl :connect `cat .nrepl-port`
Connecting to nREPL at 127.0.0.1:55432
REPL-y 0.2.1
Clojure 1.5.1
  Docs: (doc function-name-here)
        (find-doc "part-of-name-here")
  Source: (source function-name-here)
  Javadoc: (javadoc java-object-or-class-here)
  Exit: Control+D or (exit) or (quit)
```

user=>

我们的 REPL 已经准备好了。

缓存基础

`Immutant` 可以拥有任意多个缓存，每个缓存有一个名字。多个缓存工作起来有些像命名空间，将你的数据分离开来。它们可以有分别地配置用来设置如过期时间和持续时间等项。

我们可以通过 `create` 创建一个新缓存：

```
user=> (require '[immutant.cache :as cache])
user=> (def c (cache/create "testing" :ttl [1 :minute]))
{}
```

这里的参数通过向量值和其描述关键字的形式控制了默认生存时间（Time To Live, TTL）。我们加到缓存里的任何数据都会在 TTL 结束的时候过期和消失。

我们可以通过 `put` 将数据写入缓存。从缓存里读取数据和从 `Clojure` 映射里读取数据是一样的。通过这些操作，我们可以在缓存里添加一个新条目并监测它过期：

```
❶ user=> (cache/put c :url "http://pragprog.com/")
nil

❷ user=> (:url c) "http://pragprog.com/"
;; wait more than a minute

❸ user=> (:url c)
nil
```


- ❶ 我们使用 `put` 命令在缓存中添加键值对, 注意 `put` 返回的是我们上一次储存的值。
- ❷ 访问后立刻就可以返回这个值。
- ❸ 值过期后再访问会返回 `nil`。

还有很多向缓存中添加条目的方法。`put-all` 可以接受一个映射并将所有键值对放入缓存。`put-if-absent` 只会将键不存在缓存里的键值对放入缓存。`put-if-present` 只会将已存在的键对应的键值对放入缓存。`put-if-replace` 接受一个键和两个值, 一个是之前的值, 一个是新的值, 如果之前的值匹配缓存中存储的值, 就修改缓存中键对应的值为新的值, 这是一个比较和交换的操作。所有这些条件 `put` 函数都是原子操作。

在 REPL 中很容易观察到这些条件 `put` 函数是怎么工作的:

- ❶ `user=> (cache/put c :foo "foo" {:ttl -1})`
`nil`
- ❷ `user=> (cache/put-if-absent c :foo "bar")`
`"foo"`
`user=> (:foo c)`
`"foo"`
- ❸ `user=> (cache/put-if-absent c :bar "bar")`
`nil`
`user=> (:bar c)`
`"bar"`
- ❹ `user=> (cache/put-if-present c :foo "bar")`
`"foo"`
`user=> (:foo c)`
`"bar"`
- ❺ `user=> (cache/put-if-present c :baz "baz")`
`nil`
`user=> (:baz c)`
`nil`
- ❻ `user=> (cache/put-if-replace c :foo "foo" "quux")`
`false`
`user=> (:foo c)`
`"bar"`
- ❼ `user=> (cache/put-if-replace c :foo "bar" "quux")`
`true`
`user=> (:foo c)`
`"quux"`

❶ 我们在缓存中创建一个新的条目, 将默认的 TTL 覆盖成新的值, 负数会禁用过期机制。

- ② 因为:foo 已经在缓存中有值了, put-if-absent 返回了它的值并且没有做任何改动。
- ③ 因为没有:bar 的条目, 所以 put-if-absent 创建了一个。
- ④ put-if-present 检查了:foo 是否在缓存中存在, 找到它并更新了它的值。
- ⑤ :baz 不在缓存中, 所以 put-if-present 没有做任何修改。
- ⑥ put-if-replace 检查是否:foo 目前在缓存的值等于"foo"。因为它的值实际上是"bar", 所以它返回否并且没有修改值。
- ⑦ 当值匹配时, put-if-replace 返回 true 并且更新了缓存中的值。

你可以使用 delete 从缓存中删除条目, 它会返回储存在缓存中的值。

```
user=> (cache/delete c :foo)
"quux"
```

你可以传给 delete 一个值, 这样键只有在值相等的时候才会被移除。

```
user=> (cache/put c :foo "foo" {:ttl -1})
user=> nil
user=> (cache/delete c :foo "bar")
false
user=> (:foo c)
"foo"
```

这些基本的操作几乎已经覆盖了所有你对缓存会做的事情。并且, 你可以将任何 Clojure 数据直接放入缓存中。Immutant 的分布式缓存已经十分强大和方便了, 现在它更加强劲了。

记忆化

引用透明性特性是函数式编程和不可变数据结构中很美妙的事情。引用透明性的意思是针对同样的输入, 函数会产生同样的输出。这意味着没有全局状态被用来产生输出值, 也意味着没有函数带来的副作用。

数学上的函数是引用透明的, 如 4 的开根号永远是 2。一个不是引用透明的函数的例子是 (java.util.Date.)。它虽然没有参数, 但通常每次调用都会产生不同的值。

如果我们知道一个函数是引用透明的, 就可以将它的结果缓存起来。因为我们知道当输入相同时, 它的值永远不变。缓存的结果值和再次执行函数的结果一样。缓存

函数结果并在下次调用函数时使用缓存值叫作记忆化。记忆化十分常用，Immutant 提供了 memo 函数来让它便于使用。

你可以使用一下 memo：通过输入构造一个键；在缓存中搜索这个键；如果找到了，就返回缓存中的值；如果没有的话，就输入这个键执行真正的函数，然后保存结果到缓存中并返回它。让我们来看一个例子：

```

❶ user=> (defn slow-double [x]
           #=> (Thread/sleep 5000)
           #=> (+ x x))

❷ user=> (slow-double 10)
;; wait 5 seconds
20
user=> (slow-double 10)
;; wait 5 seconds
20

❸ user=> (def cached-double (cache/memo slow-double "slow-double"))
❹ user=> (cached-double 10) ;; wait 5 seconds
20
user=> (cached-double 10)
20

```

❶ 首先，我们定义 slow-double，它会在产生结果前休眠五秒钟。

❷ 调用 slow-double 十分慢，每次获得结果都要花费很长时间。

❸ 我们使用 memo 来记忆函数。memo 返回一个新的函数，将这个很慢的函数包装起来，新的函数会负责检查缓存并返回值。memo 会接受待记忆函数、缓存名称和与 create 相同的参数。这里我们就用默认的缓存参数。

❹ 第一次对 cached-double 的调用会占用五秒钟，因为输入值还没有存储在缓存中。但之后对它的调用就会立即返回相同的结果。

记忆化是很好的加速应用的方式。组合使用记忆化和 TTL 设置可以让你控制当函数没有完全引用透明时，结果保存多久比较合适。例如，对于高流量的页面，计算一下多久数据会过时，再多缓存它们一会儿可能会对服务器负载产生显著性的效果。

记忆网络页面

获取网页是一项很慢的操作，因为它涉及遍历互联网。由于很多网站不会经常改

动并且 Overwatch 只会关注网站基本的运行状况而不是内容，我们可以很容易记住获取的网页来为我们的应用加速。

这样的优点在于应用其他的部分不需要知道我们做过的任何优化，也不需要知道缓存的细节。其他部分只用调用函数来获取网页，而大部分时候这个速度是很快的。

让我们创建一个 `overwatch.link` 命名空间并创建 `fetch` 函数：

```
immutant/overwatch/src/overwatch/link.clj
(ns overwatch.link
  (:require [immutant.cache :as cache]
             [clj-http.client :as http]))
(defn fetch-slow [url]
  (http/get url))
(def fetch (cache/memo fetch-slow "fetch" :ttl 1 :units :days))
```

这个网站会每天缓存一次，这一天第一次之后的所有请求都会立即返回缓存结果。我们会用这个快速获取函数作为明天 Overwatch 其他部分的内容之一。

7.2.4 计划任务

很多任务需要在指定的时间发生。一个清理任务可能需要每天晚上运行，一个提醒可能需要几天后发给用户，或者可能需要每周生成报告。我们的应用 Overwatch 会需要定期监视 URL，而不是在响应用户请求的时候才去执行。

计划任务被集成在系统其他部分中。你的应用可以在任何时间创建任务，并且当你的应用没有部署时，任务也会暂停执行。如果你在集群上运行任务，Immutant 会保证集群上只有一个任务实例在运行。

想创建一个计划任务，Immutant 提供 `immutant.jobs` 命名空间下的 `schedule` 方法。让我们看一些例子：

```
user=> (require '[immutant.jobs :as jobs])

❶ user=> (jobs/schedule "say.hello" #(println "Hello!") :in 3000)

❷ user=> (jobs/schedule "repeat.hello" #(println "Hello!")
  #_=> :in 5000 :every 2000 :repeat 4)

❸ user=> (jobs/schedule "weekly.hello" #(println "Hello!") "0 15 8 ? * 2")
```


① `schedule` 接受的参数有任务名称,任务执行时会运行的函数和描述任务什么时候应该执行的关键词。这里我们会在三秒钟后打印一次“Hello!”。

② 这里的参数会更复杂一些。每两秒钟运行一次这个任务,每次重复四遍。第一次运行是5秒钟后开始。你可以在 Immutant 应用服务器控制台查看运行结果。

③ 除了传递一组参数,你还可以使用 `cron` 规范¹。这个任务每个星期一(第六个位置,数字2是星期的第二天的意思)上午八点十五分运行。

取消计划任务可以通过将传入 `schedule` 的任务名传入 `unschedule` 来完成。例如,想移除上一个例子的最后一个任务,就调用(`jobs/unschedule "weekly.hello"`)。

任务在这里就都介绍完了,这部分很简单但不可或缺。一旦你的应用服务器有了计划任务的功能,就很难再离开它。当然你也可以使用 `cron` 和其他类似工具,但是让 Immutant 来处理这些繁重的工作会方便很多,例如在多台机器同步 `cron` 的配置和保证正确的代码一直在运行或是运行一次后不应该再运行这种边界情况。

7.2.5 我们在第1天学到的

今天我们把 Immutant 搭建并运行起来了。我们安装和启动了一个 Immutant 应用服务器并在上面部署了一些应用,通过这些探索了 Immutant 的功能。

首先,我们创建了一个简单的“Hello, World”应用来学习怎样使用 Immutant 网络服务。因为 Immutant 是使用 Ring 来强化它的网络服务的,因此所有第4章 Ring 上的工具和技巧都可以用来构建强大的应用。

接下来我们了解了 Immutant 的分布式缓存功能,这是 Immutant 为你的应用整合的很多企业级特性之一。现今缓存已被很多网络应用产品广泛使用,但与应用服务器的紧密集成会让使用的过程更加愉悦。

最后,我们学习了怎样创建需要在特定时间运行或无限重复的任务。

现在你可能发现 Immutant 充满了这些附加的服务,很多网络应用最终都会需要这些服务,但很少有框架会直接提供。使用 Immutant,你会减少很多对额外服务的需

¹ <http://en.wikipedia.org/wiki/Cron>

求来让你的应用工作起来，所以你可以专注在业务逻辑上而不是服务关联上。

第1天的自学

查阅

- Immutant 关于缓存和任务的教程。
- Immutant 关于缓存和任务的文档。

实践

- 试用文档里其他缓存参数。使用 REPL 来学习 `:idle` 和 `:persist`。
- 计划是动态的，你甚至可以在任务中计划任务。创建一个任务，一天后运行一次，一周后运行一次，一个月后运行一次。这样的任务对增加时间间隔发送提醒会很有用。
- Immutant 可以运行任何 Ring 应用。将 Ring 章节的 Zap 移到 Immutant 来，思考一下 Immutant 提供的哪些服务是可以加入进来的。

7.3 第2天：构建数据管道

现代网络应用通常使用第三方消息队列服务来处理数据的异步和横向扩展问题。是否发送缩放和转换过的图片还是只发送邮件，队列有很多用途并且对保持你的应用与其他应用之间的松耦合性有很大的帮助。

Immutant 的消息机制就是为应用服务器构建的。不需要维护单独的系统，数据也不需要与其他协议来回传递。因为工具链已经整合进来并且可以处理大部分 Clojure 数据，使用起来十分简单。

今天我们会学习 Immutant 的消息队列和构建在其之上的抽象。

7.3.1 消息队列

消息队列是现代应用大量使用的一个简单的功能，通常被用来异步计划任务或并行处理进行中的任务。例如，从网络应用发送邮件通常会涉及将消息放入队列。

如果你在发送大量的邮件，就可能有多台机器从队列里取得任务来派发消息到远程的目的地。

通常，被添加到队列里的项目会被某些任务函数或进程移除。对于一个特定的条目，只能有一个任务来使用。这里有另外一个常见操作模式叫作发布订阅模式，被添加到队列里的项目会被所有监视这个队列的工作者收到。在 Immutant 里，它们叫作主题。

用 `lein new messaging` 创建一个叫 `messaging` 的新应用，并且编辑 `project.clj` 来添加 REPL 支持：

```
immutant/messaging/project.clj
(defproject messaging "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.5.1"]
                 [compojure "1.1.5"]]
  :immutant {:nrepl-port 0})
```

或者你可以添加 `:immutant {:nrepl-port 0}` 映射条目到你的用户设置 `~/.lein/profiles.clj` 里，这样会为你所有的 Immutant 项目添加 nREPL 支持并防止你不小心将开启着 REPL 的产品部署出去，假设你没有用 `:user` 设置来部署。下面是一个你可以使用的 `profiles.clj` 例子：

```
{:user {:immutant {:nrepl-port 0}}}
```

使用 `lein Immutant deploy` 部署应用，并使用 `lein repl :connect 'cat .nrepl-port'` 来连接 REPL。

队列

让我们来创建我们的首个队列：

```
user=> (require '[immutant.messaging :as msg])
❶ user=> (msg/start "queue.test")
❷ user=> (msg/start "queue.test")
```

❶ `start` 函数声明了一个队列或一个主题。如果名字里含有“queue”则是一个队列，如果含有“topic”则是一个主题。

❷ 对 `start` 的调用是幂等的。如果队列已经存在，再次声明这个队列没有任何效果。每个 Immutant 服务器上的应用都可以声明它使用的队列而不用担心是否其他应

用已经声明了。

发布和接受消息可以通过 `publish` 和 `receive` 来完成。我们可以发布任何 Clojure 数据并从接收端接收到。

```

❶ user=> (msg/publish "queue.test" "http://pragprog.com/")
      user=> (msg/publish "queue.test" ["http://immutant.org" "http://clojure.org"])

❷ user=> (msg/receive "queue.test")
      "http://pragprog.com"

      user=> (msg/receive "queue.test")
      ["http://immutant.org" "http://clojure.org"]

❸ user=> (msg/receive "queue.test" :timeout 5000)
      nil

```

❶ `publish` 接受队列名称和消息作为参数，并添加消息到队列中。大部分 Clojure 数据可以被发布成一条消息，这里我们发布了一个字符串和一个字符串向量。另一个常用的选择是发布映射。

❷ 你可以通过调用 `receive` 传入队列名称作为参数来接受一个消息，如果队列为空就一直等到消息到达。我们获取的就是我们发布的消息，顺序也是一样的。

❸ 我们传入 `:timeout` 选项参数给 `receive`，如果到了过期时间还没有消息到达就会返回空 (`nil`)。这里我们等五秒钟来接收消息。

我们的例子已经发布了消息并立刻收到了消息。在真实的应用中，这里会有很多接收者在等待着队列中的消息，而队列对于任务扮演着分发机制的角色。

`Immutant` 还为处理一个队列中所有消息提供了方便的函数叫作 `listen`。它接受一个函数参数来处理消息并可以并行处理消息：

```

user=> (msg/start "queue.listen")
user=> (msg/listen "queue.listen" #(println "i heard:" %))
user=> (msg/publish "queue.listen" [1 2 3])
user=> (msg/publish "queue.listen" #{:a :b :c})

```

我们创建一个新的队列并搭建一个监听者来打印监听到的所有消息。发布两个消息后，你可以检查运行中的 `Immutant` 服务器控制台输出来查看结果。应该会看到类似如下输出：


```
22:13:26,581 INFO [stdout] (Thread-16 (HornetQ-client-...)) i heard: [1 2 3]
22:16:48,023 INFO [stdout] (Thread-16 (HornetQ-client-...)) i heard: #{:a :c :b}
```

队列十分简单：某组任务发布消息到队列中，另外一组任务接收这些消息并做些操作。任何消息只会被一个任务接收到。消息发布者不需要知道接受者的任何信息，无论接受者是谁、有多少或是否接收了消息。只要知道队列的名称了解这些消息就够了。这使得组件之间达到了松耦合的状态。

主题则有些不同，接下来我们看一下。

主题

主题是一个广播机制。消息被发送到主题后，所有的监听者会收到消息的一份拷贝。如果你有几个任务处理者并且需要在它们之间协调一些行为，如设置一个新的配置或关闭，那主题是很适合的。

在 Immutant 里，主题和队列有几乎一样的 API；区别是发送给主题的消息会被所有接收者和监听者接收到，而不仅仅是其中的一个。我们可以同时创建主题和队列，附上一些监听者，然后发布一些消息给它们，这样可以清楚地看到主题和队列的区别。让我们从队列开始：

```
❶ user=> (msg/start "queue.multi")
❷ user=> (defn worker [m]
  #_=> (let [id (.getId (Thread/currentThread))]
    #_=> (println "worker" id ":" m)
    #_=> (Thread/sleep 5000)))
❸ user=> (msg/listen "queue.multi" worker :concurrency 5)
❹ user=> (dotimes [i 3] (msg/publish "queue.multi" (str "hello " i)))
```

❶ 首先，我们创建了一个新的队列。

❷ 我们创建了一个简单的工作者来打印消息及其线程 ID 并休眠一会儿，模拟执行耗时长任务的状态。

❸ 我们给队列附上一个监听者，并发量设置为 5，表示允许最多五个线程处理这些消息。

❹ 三条不同的消息被发布到队列中。你可以检查 Immutant 服务器控制台来看这

个测试的结果。应该有类似如下的结果：

```
23:14:59,247 INFO [stdout] (Thread-25 (HornetQ-client-...)) worker 241 : hello 0
23:14:59,252 INFO [stdout] (Thread-20 (HornetQ-client-...)) worker 225 : hello 1
23:14:59,260 INFO [stdout] (Thread-26 (HornetQ-client-...)) worker 242 : hello 2
```

3个不同的线程处理了消息，但每个消息只被处理了一次。我们再对主题做同样的实验来看看：

```
user=> (msg/start "topic.multi")
user=> (msg/listen "topic.multi" worker :concurrency 5)
user=> (dotimes [i 3] (msg/publish "topic.multi" (str "hello " i)))
```

现在观察输出，你会看到十分不同的结果：

```
... worker 250 : hello 0
... workerworker worker 254249 :251: : workerhello 0hello 0
...
... 248 hello 0: hello 0
...
... worker 249worker :254 : hello 1hello 1
...
... worker worker251 worker250: :hello 1 248
... hello 1
... : hello 1
... worker 249 : hello 2
... worker 250 : hello 2
... workerworker 254 248: hello 2:
... workerhello 2
... 251 : hello 2
```

输出是所有线程交错的，你可以看到每个消息被处理了五次，每个并发的监听者一次。

通过队列和主题让很多有趣的架构成为可能，并且这些架构可以经常在不同的组件中达到降低耦合度的效果，以此可被扩展到大型的工作负载中。

7.3.2 管道

一些任务会涉及多步骤，每个阶段接受一些输入并进行计算，为下一个阶段生成一些结果。很多计算可能需要并行运行来跟上输入的速度。

为了解决这类问题，你可能会让每个阶段从队列中读取输入，做自己的计算，然

后把结果发布成一个新的队列。因为这是一个常用模式，Immutant 提供了管道，将架构包装成一种更方便使用的形式。

管道使这类分发和并行数据流看上去和 Clojure 自己的数据流很像。考虑一下这些负责发送邮件的常用 Clojure 代码：

```
immutant/messaging/src/messaging/email.clj
(def send-email [username]
  (-> username
    get-email
    make-msg
    send-msg))
```

这段代码从某些地方如数据库获取了用户邮件地址，创建了邮件消息，并发送出去，假设函数 `get-email`、`make-msg` 和 `send-msg` 已经存在。

通常的情况是这些函数需要访问其他系统，如 `get-email` 和 `send-email`，整个过程要花费很长时间。你的应用生成的其他页面可能不需要依赖于这些任务的输出，所以希望这部分工作可以异步执行。

通过 Immutant，你可以把同样的代码通过 `pipeline` 函数转换成管道：

```
immutant/messaging/src/messaging/email.clj
(require '[immutant.pipeline :as pl])

(def send-email-pipe
  ① (pl/pipeline "email"
    ②   get-email
    make-msg
    send-msg
    ③   :concurrency 5))
```

① `pipeline` 函数接受管道名称、步骤和可选项作为参数。它会返回一个函数，将数据放入我们指定给 `send-email-pipe` 的管道的第一步。

② 这里的函数和在 Clojure 例子里用 `->` 列出的函数一样。对于每个函数，`pipeline` 会在管道中创建一个步骤。基于 `:concurrency` 参数的设置，每个步骤的若干拷贝会同时运行。记住，在这些步骤之间，步骤的结果会被发布成管道里的消息，有很多工作者在监听这些消息并等着执行下一步。

③ `pipeline` 的最后一个参数会被应用到管道的所有步骤上。这里我们设置并发量

为5，意味着每一步会有五个并发的工作者处理消息。

给管道发送数据就像调用管道方法一样简单。例如，(`send-email-pipe "someone"`)会完成所有事情。管道会返回一个延迟，这是一个对未来的值的引用。你可以通过调用它的 `deref` 或添加前缀 `@` 得到实际的值，这个值一开始可能是阻塞的直到可用时才能得到。在当前这个管道中，返回值不是十分有用，你可以通过传入一个值为-1的 `:result-ttl` 参数来禁用对返回值的存储。默认的，是会保存一个小时。

我们也可以通过 `step` 函数针对每一步更改参数：

```
immutant/messaging/src/messaging/email.clj
(def send-email-pipe2
  (pl/pipeline "email2"
    ① (pl/step get-email :concurrency 10)
      make-msg
      send-msg
      :concurrency 5))
```

① `step` 函数让我们可以在这一步覆盖 `:concurrency` 参数。也许 `get-email` 很慢，需要更多的工作者来满足需求。

到这里就学完管道的基本内容了。管道可以调用其他管道，并且输入值可以被发送到管道的任何步骤。在为 `Overwatch` 构建管道来处理 URL 的过程中，我们将会看到更多高级的使用方式。

7.3.3 Overwatch 的管道

让我们想象一下 `Overwatch` 会怎样为监视器处理 URL。我们可以发布 URL 到一个队列中。一组并行的工作者可以监听这个队列并取得 URL 的内容，将结果发布到一个新的队列中。另一组工作者可以获取到内容并解析出其中的 URL。这些新的 URL 会被传递给另一组工作者来获取内容，然后最后一组工作者会将结果储存到数据库。图 7-1 展示了这个管道的流程。

在管道的每一步里，结果会被收集并放入队列中供下一步使用。你可能已注意到，解析链接的步骤看上去有些不同。网页含有很多链接，每个链接会被作为它的任务放入下一个队列。这种对结果的扩散方式更平均地分配了负载。

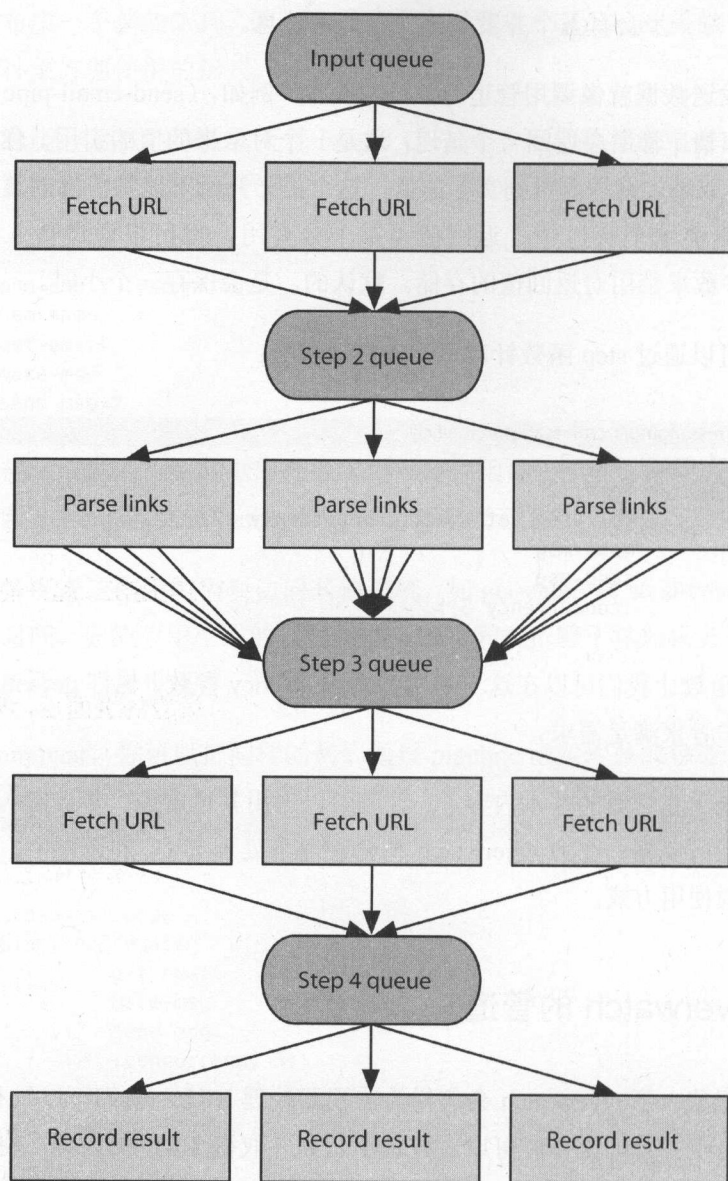


图 7-1 Overwatch 管道

通过 Immutant 可以很轻松地创建这样的管道。从这个 `overwatch.check` 命名空间开始，我们要构建我们的管道了：

```
immutant/overwatch/src/overwatch/check.clj
```

```
(ns overwatch.check
```

```
  ① (:require [immutant.pipeline :as pl]
```

```

[immutant.cache :as cache]
[clj-http.client :as http]
[net.cgrand.enlive-html :as html]
[clojure.tools.logging :as log])
(import [java.io StringReader]
        [java.net URL]))

```

```

2 (def check-url
  (pl/pipeline :check-url
    fetch-url
    parse-links
3   fetch-url
    record-result
    :concurrency 5
    :result-ttl -1))

```

① 稍后当我们在实现管道的步骤时，会需要这些输入。

② 管道的每一步都试图去做一件简单的事情。将它们串连在一起并使用 `Immutant` 来并行每一步，这个管道就可以完成大量的工作。

③ `fetch-url` 在管道中再次出现是因为在解析了链接对应的页面后，我们希望获取所有找到的新链接。

让我们来实现这个管道的第一个函数 `fetch-url`，它会出现在这一步和第三步。这段代码和这个例子里的其他代码应该处于 `check.clj` 文件的 `ns` 代码块之后，`check-url` 定义之前。

```

immutant/overwatch/src/overwatch/check.clj
(defn fetch-url [url]
  {:url url
   :response (http/get url {:throw-exceptions false})})

```

`fetch-url` 只是返回了一个储存 URL 和提取出来的内容的字典。通常 `clj-http` 会抛出 400-和 500-级别状态码的异常，但是我们已经关闭了这一功能并替换成获取正常的响应映射。

下一步是 `parse-links`，这里必须获取响应的正文，提取出所有的 `<a>` 标签并发送出去以供获取。这一步是最复杂的，因为我们必须规范化出现在页面大量的不同表单下的所有链接，如 `http://example.com/foo`、`/foo` 和 `../bar`：

```

immutant/overwatch/src/overwatch/check.clj
1 (defn normalize-url [base-url url]
  (cond
    2 (.startsWith url "http") url
    3 (.startsWith url "/" )
      (let [base-url (URL. base-url)
            new-url (URL. (.getProtocol base-url)
                          (.getHost base-url)
                          (.getPort base-url)
                          url)]
        (str new-url))
    4 :else
      (let [base (re-find #".*/?" base-url)
            base (if (.endsWith "/" base) base (str base "/"))]
        (str base url))))

```

❶ normalize-url 接受基 URL 和一些标签的 href 属性值作为参数。这里有三种情况需要处理：包括网络协议的 URL 全路径、绝对路径和相对路径。

❷ 如果 URL 以 http 开始，我们假设它是一个全路径，不需要任何转换。

❸ 对于绝对路径，我们使用基 URL 的网络协议，主机和端口加上新的路径构建一个新的 URL。

❹ 相对路径复杂一些。我们通过正则表达式去掉基 URL 最后一个斜线之后的所有字符，然后拼上这个相对路径。

现在我们可以规范化所有 href 属性，可以创建 parse-links 了：

```

immutant/overwatch/src/overwatch/check.clj
1 (def fetch-cache
  (cache/cache "fetch" :ttl 1 :units :days))
2 (def link-cache
  (cache/cache "links" :ttl 2 :units :days))

3 (defn parse-links [data]
  (let [url (:url data)
        response (:response data)]
    4 nodes (html/html-resource (StringReader. (:body response)))
    5 links (->> (html/select nodes [[:a (html/attr? :href)]])
    6 (map (comp #(normalize-url url %) :href :attrs))
    (apply hash-set url)))

  (log/info "URL:" url links)
  7 (cache/put fetch-cache url response)
  (cache/put link-cache url links)
  8 (pl/fanout links)))

```

❶ `fetch-cache` 会存储获取所有 URL 的响应。你可能注意到它的名字和昨天 `fetch` 函数的缓存名称一样，这意味着它们使用相同的缓存。我们这里储存的响应对 `fetch` 是可用的。

❷ `link-cache` 会将 URL 和我们发现的其所有链接储存在一起。这里设置了一个 URL 和它对应的页面之间的关系。

❸ 我们使用 `Enlive`（在第 4 章 `Ring` 可以获得 `Enlive` 更多内容）来解析链接。首先，我们将响应正文转化成一个可以操作的结点列表。记住 `Clojure` 的这个线程末尾操作符 `->>`，它会获取每一步的结果并将其作为下一步函数的最后一个参数传入。

❹ 在这些结点中，我们选择出含有 `href` 属性的 `<a>` 标签。

❺ 规范化我们选择的所有链接，有些函数式编程的感觉。`map` 使用 `comp` 创建的函数转化了每个结点。这个函数是选择所有属性 `(:attrs)`，返回 `href` 属性 `(:href)` 然后规范化值的组合。

❻ 在所有的链接上运行 `hash-set` 会将可能存在重复链接的列表转化成没有重复的链接集合。

❼ 任务完成后，我们将主 URL 的响应储存在 `fetch-cache` 上，将链接的列表储存在 `link-cache` 上，以供其他代码将来使用。

❽ `fanout` 接收列表中的每一项并将其发送到队列中，供管道的下一步使用。这意味着每个 URL 输入可能会为下一步生成很多个输出。注意如果我们把这些链接放入管道的第一步，就创建了一个网络爬虫。

我们管道的最后一步用来记录结果：

```
immutant/overwatch/src/overwatch/check.clj
(defn record-result [data]
  (let [url (:url data)
        response (:response data)]
    (log/info "Link:" url (:status response))
    (cache/put fetch-cache url response)))
```

这个函数将响应写入 `fetch-cache` 供其他代码使用。

现在我们构建了自己的管道，我们可以通过 `lein immutant deploy` 部署应用并通

过 REPL 连接上进行测试。大概会打印出如下输出：

```
user=> (require '[overwatch.check :as check])
nil
user=> (check/check-url "https://pragprog.com/")
#<Delay@574b02: :pending>
user=> URL: https://pragprog.com/ #{...}
Link: https://pragprog.com/terms-of-use 200
Link: https://pragprog.com/about 200
Link: http://www.defectivebydesign.org/drm-free 200
Link: http://pragprog.com/news/test-ios-apps-with-ui-auto... 200
Link: https://forums.pragprog.com 200
Link: https://pragprog.com/frequently-asked-questions/returns 200
Link: http://pragprog.com/news/no-batteries-required-book-sale... 200
Link: https://pragprog.com/resources/credits 200
Link: https://pragprog.com/my_profile 200
Link: http://pragmaticstudio.com 200
<<omitted output>>
```

通过一些简单的函数和 Immutant 的管道，我们创建了一个可以在多个机器和集群核之间横向扩展的高并发数据流。我们不仅使用了 Immutant 内置的消息队列，还了解到它负责所有的细节问题。

7.3.4 我们在第2天学到的

消息机制是很多现代应用十分重要的部分，以至于很多创业公司的存在只是为了向开发人员提供简单的消息机制功能。今天我们花了一整天来探索 Immutant 的内置消息机制支持。

首先，我们从队列开始。数据被发布到队列中，之后被潜在的很多接收者之一获取。这非常适合移交任务给大量的工作者或者为你的应用中不需要互相了解的组件解耦。

我们还学习了主题，它类似于队列，你可以发布消息到上面。但是和队列不一样的是，消息会被所有接收者收到，而不是仅仅一个。主题很适合发布全局状态或构建聊天系统。主题是一个发布-订阅系统的例子，并且是很强大有用的工具。

最后，我们通过 Immutant 管道构建了 Overwatch 链接监视器的基本功能。管道是基于队列之上的一个方便的抽象，它在步骤队列间组织创建数据流。每个步骤会处理从一个队列来的输入并将结果自动发送给其他队列，而且可以轻松地并行所有步骤，来创建可以处理大数据量的管道。

第2天的自学

查阅

- Immutant 关于消息传递的指南。
- 消息传递设置参数文档，例如:priority 和:error-handler。

实践

- 主题对于构建聊天环境类的功能真的是很好的抽象。想想你怎样通过主题来设计像 Twitter 这样的应用。你能在 REPL 上构建一个简单的聊天系统吗？
- 你可以在管道的任何一步插入数据，甚至在相同管道的其他步骤。自行尝试直接将 URL 放入 REPL 的第二阶段 fetch-url。

7.4 第3天：多语言应用

不同的工具有不同的优劣势，所有人都愿意为每个任务使用更适合的工具。很多开发人员会将很多种编程语言混合在一起来构建应用，即多语言开发。Java 虚拟机已经使多语言编程变得前所未有的简单，Immutant 更是通过一些叠加支持这种开发形式。例如，你可以在 Ruby 里编写你的主网络应用，来提交数据供 Immutant 的管道处理和接收结果。或者可能你的网络应用需要使用一个规则引擎；Java 生态系统里有很多，只需要一个函数调用就好了。

7.4.1 叠加

Immutant 叠加允许将不同的语言混合进同一个应用服务器。目前，Immutant 支持在 Immutant 上叠加 TorqueBox，可以让你将 Clojure 和 Ruby 应用混合匹配起来¹。

TorqueBox 和 Immutant 应用共享对底层 JBoss 平台同样的访问和功能，包括缓存、消息机制和任务。Immutant 应用可以将消息发布到队列上供 TorqueBox 应用监听，TorqueBox 缓存的数据 Immutant 也可以访问。

¹ <http://torquebox.org/>

创建一个叠加

为了搭建 TorqueBox 叠加，你需要运行 `lein Immutant overlay`，这会下载并将 TorqueBox 与我们的 Immutant 服务器安装到一起。安装完成之后，如果应用服务器仍在运行，你需要按[Ctrl-C]组合键关闭，然后重新启动：

当 Immutant 启动后会输出正在运行的 Immutant 和 TorqueBox 的版本。如果一切工作正常，你会看到类似如下的输出：

```
$ lein immutant overlay
No feature set provided, assuming 'torquebox'
Downloading http://downloads.immutant.org/.../torquebox-dist-bin.zip
<<omitted output>>
$ lein immutant run
Starting Immutant: /Users/jack/.immutant/current/jboss/bin/standalone.sh
=====
<<omitted output>>
22:07:30,922 INFO ... Welcome to TorqueBox AS - http://torquebox.org/
22:07:30,922 INFO ... version..... 3.x.incremental.1728
<<omitted output>>
22:07:30,923 INFO ... Welcome to Immutant AS - http://immutant.org/
22:07:30,923 INFO ... version..... 1.0.1 (PuntoBueno)
<<omitted output>>
```

现在你同时有 Ruby 和 Clojure 应用服务器了，让我们来看看可以做什么。

TorqueBox 中的 Sinatra

首先，我们会通过 Sinatra 创建一个 TorqueBox 应用，第 1 章第 1 页 Sinatra 已经介绍过细节了。创建一个新的项目目录叫 `rubypoly`：

```
$ mkdir rubypoly
$ cd rubypoly
```

然后，我们会通过创建 `config.ru`、`hello.rb`、`Gemfile` 和 TorqueBox 配置文件 `config/torquebox.rb` 来创建一个“Hello, World”应用。

```
immutant/rubypoly/config.ru
require 'torquebox'
require './hello'
```

```
run Sinatra::Application
```

```
immutant/rubypoly/hello.rb
require 'sinatra'
```

```
get '/' do
  "Hello, world!"
end
```

```
immutable/rubypoly/Gemfile
source 'https://rubygems.org'
gem 'torquebox'
gem 'sinatra'
```

```
immutable/rubypoly/config/torquebox.rb
```

```
1 TorqueBox.configure do
2   web do
     context "/rubypoly"
   end
end
```

❶ TorqueBox.configure 代码段在我们的应用中配置了不同的 TorqueBox 服务，相当于 Immutable 应用的 `immutable/init.clj`。

❷ 我们将网络服务配置的上下文设置成我们自己应用的名称。与 Immutable 不同，TorqueBox 会默认将其设置成 “/”。

应用创建后，我们需要搭建环境，安装依赖并将其部署。像 Immutable 应用通过 Leiningen 部署一样，TorqueBox 应用通过 `torquebox` 命令部署。

```
$ export TORQUEBOX_HOME=$HOME/.immutable/current
$ export PATH=$TORQUEBOX_HOME/jruby/bin:$PATH
❶ $ bundle install
Using rake (10.0.3)
Using blankslate (2.1.2.4)
Using parslet (1.4.0)
Using edn (1.0.0)
<<omitted output>>
❷ $ torquebox deploy
Deployed: rubypoly-knob.yml
into: /Users/jack/.immutable/current/jboss/standalone/deployments
```

❶ 我们用 JRuby 的 `bundle` 命令来安装我们列在 Gemfile 里的依赖。这会下载并安装任何缺失的 Ruby 打包库。

❷ `torquebox deploy` 会将应用部署到覆盖在 Immutable 之上的 TorqueBox 应用服务器上。这相当于 Clojure 应用的 `lein Immutable deploy`。

应用部署后，你会在 `http://localhost:8080/rubypoly/` 看到熟悉的“你好，世界”消息。

多语言消息机制

消息队列和主题是对应用服务器上运行的所有应用可用的资源。你可以在 Immutable 应用发布的 TorqueBox 应用中轻松地使用消息。消息机制服务不仅将应用中的代码解耦，每部分实现的代码还可以在不同的应用中，甚至使用不同的语言。

让我们为 rubypoly 添加一个消息处理程序，来看看它们是怎么工作的。首先，添加以下代码到 config/torquebox.rb 中 Web 部分之后，configure 块之内：

```

immutable/rubypoly/config/torquebox.rb
1 options_for :messaging, :default_message_encoding => :edn

2 topic "topic.poly" do
3   processor PrintProcessor
end
  
```

❶ 缺省情况下，TorqueBox 期望消息是序列化的 Ruby 对象，但是 Immutable 使用的是可扩展数据符号（extensible data notation, EDN）。TorqueBox 应用也可以设置使用 EDN。你还可以设置 Immutable 应用和 TorqueBox 应用使用队列或主题时用 JSON 格式，但是我们一会儿会看到，EDN 有一些独特的优势。

❷ 这里是对主题的设置，大体上相当于 Immutable 的 `immutable.messaging/start` 函数。任何针对主题的设置都放在这个代码块里。

❸ 监听者在 TorqueBox 里是 `MessageProcessor` 子类的实例，而不是简单的函数，`processor` 指令告诉 TorqueBox 使用哪个子类。

我们还需要实现这个 `PrintProcessor` 类：

```

immutable/rubypoly/print_processor.rb
1 class PrintProcessor < TorqueBox::Messaging::MessageProcessor
2   def on_message(msg)
      puts "ruby says: " + msg.to_s
    end
end
  
```

❶ 处理器子类 `MessageProcessor`。这里是通过 `on_message` 完成的。

❷ `on_message` 接受消息并可以完成任何需要的处理。当使用 EDN 编码时，传入的消息会被转换成一个合适的 Ruby 对象。我们的处理器只是将消息打印成字符串，并附上表明是来自于 Ruby 应用的说明。

我们的 rubypoly 应用已经准备好重新部署了。运行 `torquebox deploy`，并创建一个 Clojure 应用来与之互动。

运行 `lein new clojurepoly` 来创建一个新的 Immutant 项目用来测试，编辑 `project.clj` 匹配如下代码：

```
immutant/clojurepoly/project.clj
(defproject clojurepoly "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.5.1"]]
  :immutant {:nrepl-port 0})
```

使用 `lein immutant deploy` 来部署应用，并检查应用服务器输出内容里的 nREPL 端口号。使用 `lein repl :connect 'cat .nrepl-port'` 来将 REPL 连接到 clojurepoly 应用。

创建一个主题并附上一个简单的监听者：

```
user=> (require '[immutant.messaging :as msg])
user=> (msg/start "topic.poly")
user=> (msg/listen "topic.poly" #(println "clojure says: " %))
```

现在 Ruby 和 Clojure 应用都在监听同一个主题 `topic.poly`。发布一个消息到这个主题上，观察应用服务器输出：

```
user=> (msg/publish "topic.poly" "polyhello")

23:37:51,375 INFO [stdout] (...) clojure says: polyhello
23:37:51,390 INFO [stdout] (...) ruby says: polyhello
```

因为我们使用的是主题而不是队列，所有的监听者都会接收并处理消息，所以 Ruby 和 Clojure 应用都打印了日志。

因为两方用的都是 EDN 编码，我们可以在消息中传递映射、数组甚至是日期，而双方都会以本地对象的形式看到它们。

```
user=> (msg/publish "topic.poly" [:a 1 "foo"])

23:42:41,443 INFO [stdout] (...) clojure says: [:a 1 foo]
23:42:41,470 INFO [stdout] (...) ruby says: [:a, 1, "foo"]

user=> (msg/publish "topic.poly" {:a 1 :b 2})

23:44:29,321 INFO [stdout] (...) clojure says: {:a 1, :b 2}
23:44:29,345 INFO [stdout] (...) ruby says: {:a=>1, :b=>2}

user=> (msg/publish "topic.poly" (java.util.Date.))

23:45:37,183 INFO ... clojure says: #inst "2013-09-23T05:45:37.166-00:00"
23:45:37,207 INFO ... ruby says: 2013-09-23T05:45:37+00:00
```

EDN 编码使得在 Clojure 和 Ruby 之间传递数据变得十分方便。所有的序列化和反序列化都已替你完成，每一方的输入输出都是本地对象。

多语言合作不仅仅止步于此。分布式缓存也可以从不同语言的不同应用来访问。一个运行在 Torquebox 叠加层上的 Ruby 应用可以轻松地从此缓存访问我们 Overwatch 管道的结果，或自己对其做出修改。

7.4.2 集群

大量的 Immutant 结点可以通过集群连接到一起，扩大了部署应用的可用资源数量。缓存、任务、消息机制和其他 Immutant 服务是跨集群可用的。

你可以通过运行 `lein immutant run --clustered` 将 Immutant 启动成集群的一部分。不过为了示范用，想在一台机器上模拟出一个真实的集群，还是有些工作要做的。

首先，通过按下 [Ctrl-C] 组合键关闭你的应用服务器（如果它在运行着）。然后，复制 Immutant 应用服务器到另外一个位置并启动一个集群。在一个命令行窗口下执行如下命令：

```
$ cp -R ~/.immutant/current/ /tmp/immutant-node2
$ lein immutant run --clustered
```

这会启动集群的第一个结点。现在我们可以使用第二个命令行窗口去启动第二个结点：

```
$ rm -rf /tmp/immutant-node2/jboss/standalone/data
$ IMMUTANT_HOME=/tmp/immutant-node2 \
  lein immutant run --clustered \
    -Djboss.node.name=two \
    -Djboss.socket.binding.port-offset=100
```

额外参数设置了一个自定义的结点名称并且让 Immutant 去使用不同的端口，这样就不会与第一个结点相冲突。

现在我们有一个运行着的小集群了，来创建一个新的 REPL 支持的项目并把它部署到这个集群的所有结点上。运行 `lein new cluster` 并编辑 `project.clj` 为如下内容：

```
immutant/cluster/project.clj
(defproject cluster "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.5.1"]]
  :immutant {:nrepl-port 0})
```

将应用部署到所有的结点上：

```
$ lein immutant deploy
<<omitted output>>
$ IMMUTANT_HOME=/tmp/immutant-node2 lein immutant deploy
<<omitted output>>
```

准备集群的最后一步是连接一对 REPL 至其上，每个 REPL 对应一个结点。在每个结点的日志输出中找到它们的 nREPL 端口号，在各自的终端上为其启动一个 REPL 会话。

集群化的缓存

Immutant 支持很多种模式的集群缓存，但默认的是分布式。在这种模式下，每个条目在集群中被复制到两个结点中。因为这种复制，一个结点的数据丢失意味着缓存的条目还是可以在失败中存活下来的。

当你需要一个缓存条目时，数据是本地的还是远程的都没有关系；如果在集群里找到了就会返回给你，你可以轻松地在 REPL 里操作它。在 REPL 的第一个结点，运行如下代码：

```
user=> (require '[immutant.cache :as cache])
user=> (def c (cache/cache "cluster" :ttl [1 :day]))

user=> (cache/put c :immutant "http://immutant.org")
```

在第二个 REPL 里，运行如下代码：

```
user=> (require '[immutant.cache :as cache])
user=> (def c (cache/cache "cluster" :ttl [1 :day]))

user=> (:immutant c)
"http://immutant.org"
```

除此以外就不用做什么了，它已经开始工作了。我们第一天看到的所有其他缓存函数也都可以集群间工作。

集群化的消息机制

消息机制在集群上也可以如你期望的那样工作。消息在集群的接收者之间是负载均衡的，在一个结点上发布的消息可能会被不同的结点接收，或被主题情况下的所有

结点接收。没有额外的设置和调用，因为当应用服务器集群化时，消息系统处理事情是透明化的。

让我们在每个结点上都创建一个队列并附上一些监听者。一旦我们搭建好队列和监听者，就可以发布一些消息到队列上看看会发生什么。

在第一个结点的 REPL 运行如下代码：

```
user=> (require '[immutant.messaging :as msg])
user=> (msg/start "queue.cluster")
user=> (msg/listen "queue.cluster" #(println "one:" %) :concurrency 2)
```

现在在第二个结点的 REPL 上运行类似的代码，将 println 参数改成读 two：

```
user=> (require '[immutant.messaging :as msg])
user=> (msg/start "queue.cluster")
user=> (msg/listen "queue.cluster" #(println "two:" %) :concurrency 2)
```

现在在每个 REPL 上都发布一些消息：

```
user=> (msg/publish "queue.cluster" "first")
user=> (msg/publish "queue.cluster" "second")
user=> (msg/publish "queue.cluster" "third")
user=> (msg/publish "queue.cluster" "fourth")
```

检查你的两个结点的日志，看看哪个结点处理了哪些消息。应该会有类似如下的结果：

```
# on node 1
<<omitted output>>
01:07:43,115 INFO [stdout] ... one: first
01:07:49,935 INFO [stdout] ... one: third
<<omitted output>>
# on node 2
<<omitted output>>
01:07:45,934 INFO [stdout] ... two: second
01:07:53,198 INFO [stdout] ... two: fourth
<<omitted output>>
```

就像你看到的一样，Immutant 集群按照轮循的方式在两个结点间交替发布了消息。如果你正在将任务移交到队列上，通过设置适合的:concurrency 参数可以十分容易地利用到机器的每个核，以及你的集群上的每个机器。扩展工作负载已经不能再简单了。

集群化的任务

最后我们来了解一下 Immutant 集群中的任务集群。Immutant 集群上的任务默认是单例，它们只在集群上的一台机器上运行。如果一个集群的结点失败了，任务会在其他某个结点上运行。

不像缓存和消息那样，任务需要一些设置。集群为了保证一个给定的任务只在一个结点上运行，它需要将所有任务分发到所有结点上。

就像你的应用部署时，每个结点运行的初始化一样，初始化任务不是一件难事。不过，在 REPL 上的或被应用动态计划的任务会需要一些方法来在集群间计划。幸运的是，我们已经有了符合需求的工具：主题。

为了展示集群化的任务，我们要创建一个主题和一些监听者来在我们的集群上计划任务。然后我们会计划一些任务并观察它们是怎样在集群上操作的。

在每个 REPL 上运行如下代码：

```
user=> (require '[immutant.jobs :as jobs])
❶ user=> (def sched [{:keys [name code interval]}]
  #_=> (jobs/schedule name (eval code) :every interval))

#'user/sched
❷ user=> (msg/start "topic.jobs")
user=> (msg/listen "topic.jobs" sched)
```

❶ sched 函数是一个用来分解收到的消息并计划出对应的任务的助手。注意这里 eval 的使用，会将数据动态转化为代码，允许我们将函数分发往消息的集群。

❷ 这里我们创建了一个主题并附上一个监听者，直接传递消息给 sched。因为我们使用的是主题，每个监听者都会获得一份消息复制并计划这个任务。

现在，在某一结点上运行如下代码来计划一个新的任务：

```
user=> (msg/publish "topic.jobs"
  #_=> {:name "the-job"
  #_=> :code '(fn [] (println "doing a good job"))
  #_=> :interval 5000})
```

这会发送任务描述到主题上，每个结点都会收到。每个结点的监听者会传递消息

给 sched，并会用自己的结点计划这个任务。

集群某处的一个结点会开始执行这个任务。你可以看看每个结点的日志来找出是哪个结点在每五秒钟一次地重复着这个任务的消息：

```
«omitted output»
01:32:02,619 INFO [stdout] (JobScheduler$cluster.clj_Worker-1) doing a job
01:32:07,620 INFO [stdout] (JobScheduler$cluster.clj_Worker-2) doing a job
01:32:12,620 INFO [stdout] (JobScheduler$cluster.clj_Worker-3) doing a job
«omitted output»
```

集群中其他的结点不会再运行这个任务，因为 Immutant 保证了任务只会被集群中的一个结点执行。

我们现在可以在运行任务的结点上模拟结点失败，来看看 Immutant 集群是怎样反应的。在重复打印任务消息的结点上点击[Ctrl-C]组合键来关闭它，然后观察其他结点的日志输出。你会看到这个任务继续稳健地运行着，甚至可以重新启动刚刚关闭的结点，任务还是会继续运行在当前的结点上。

Immutant 的集群操作简单并且十分强大。你在一个机器上使用的所有服务都会透明化地工作在整个集群上，这就使得你的应用可以扩展到支持更大的工作负载，更稳健地去处理失败。

7.4.3 我们在第3天学到的

今天我们了解了 Immutant 的两个如此显著的功能特性：通过 Immutant 叠加进行多语言编程和在一个 Immutant 集群上运行多个结点，也就完成了我们对 Immutant 的探索。

通过在 Immutant 上叠加 Torquebox 应用服务器，我们可以把 Ruby 和 Clojure 代码混合匹配在一起，其之间的通信是通过 Immutant 的缓存和消息机制服务实现的。这是极其有用的功能，让你可以为任务使用最好的工具或使用其他语言编写的已有库。

集群化让我们可以在多结点间集中使用资源。因为添加结点使集群的可用内存大小增加了，于是缓存变大了。消息被分发出去使得任务可以传播至多个结点。并且如果故障发生，一个结点运行的任务会被优雅地转移到其他的结点上。

最棒的是，这些都是在 Immutant 中内置的，并且完全与其他系统集成在一起。

第3天的自学

查阅

- Torquebox 指南和文档。
- Immutant 其他服务的指南，如分布式事务处理和守护进程这些我们来不及了解的。

实践

- 试着使用 Immutant 的其他服务，如 Ruby 中的缓存和任务。通过 Torquebox 文档获取帮助。
- 创建一个主题和监听者来在集群上计划 Overwatch 的 check-url 任务。
- 通过添加一个网络前端界面提交任务到集群中并让用户浏览结果表单来完成 Overwatch。

7.4.4 对 Jim Crossley 的采访

Jim Crossley 是 Immutant 和 Torquebox 的主要贡献者，也是 Red Hat 的首席软件工程师。

我们：Immutant 项目是怎么开始的？将 JBoss 包装成更好的形式的想法是怎么产生的？

Jim：Torquebox 是第一个“JBoss 的优美包装”，创建于 2008 年 Bob McWhirter 从 JBoss 休假一年去学习 Ruby on Rails 之后。当他回来时，意识到 JRuby 想让他将他的新宠与雇主的旗舰产品整合到一起，从而不仅证明了自己还简化了 Ruby 应用的开发。2010 年，他成立了“Project Odd”团队，包括我和 Toby Crawley 来帮他实现这个愿景¹。因为 Torquebox 很成熟，我们想寻找另外的机会来扩展 JBoss 至多语言化。Toby 和我都认为 Clojure 十分合适，Toby 在 2011 年 9 月为 Ring 应用构建了一个 AS7

¹ <http://projectodd.org>

部署者证明了这个概念。Immutant 由此进化而来。

我们：消息机制和缓存绝对是大部分现代网络应用不可或缺的功能，提供此类功能让 Immutant 十分独特。还有什么是 Immutant 提供的但开发人员可能错过了的？

Jim：Immutant 是一个商业服务的整合，大部分重要的网络应用在其发展过程中都会需要这些服务。除了消息机制和缓存机制，Immutant 还提供了内置的计划机制、守护进程、事务和集群等。几乎很多主流应用除了数据库都需要的功能，不过如果你的数据库请求很少，也可以就使用一个持久的存储。因为这是一个整合的平台，你的应用部署时附带的复杂性就大大地降低了。一个整合栈还可以让你得到统一化的服务规模，例如只需为你的 Immutant 集群添加更多的结点就可以得到消息机制的自动负载平衡、缓存数据栅格的扩展和计划任务守护进程的高可用性。

我们：Immutant 有没有在整合除了 Ruby 以外的其他语言来扩大它的多语言范围？

Jim：Red Hat/JBoss 是全基于 JVM 多语言之上。除了 Immutant 用 Clojure 和 Torquebox 用 Ruby 之外，还有 Escalante 用 Scala，并且我们还有一些 JavaScript 开发正在进行之中。我们有 Overlay 项目可以将 Ruby 和 Clojure 应用都托管在一个单独的应用服务器上，但我们觉得还可以做得更好。我们现在正在评估将 Vert.x 作为一个统一我们多语言范围的手段，使其能被开发人员嵌入他们的应用中，就像现在很多 Clojure 开发人员嵌入 Jetty 一样。这可以让你嵌入任何商业服务到你的应用中，而不用去管它是不是基于 JVM 的语言。

我们：交互式开发是 Lisp 的重要部分，Immutant 看样子完全接受这种方式。这会改变你和你的用户创建网络应用的方式吗？

Jim：老实说，Immutant 里最让我兴奋的是 REPL。当我的应用在 Immutant 上部署运行后，我喜欢在 REPL 上增量式地构建我的应用。我喜欢实时地与集成的服务交互，为这些服务写测试并立刻运行它们而不需要任何模拟或打包或部署步骤。这是一个紧密无缝、非常流畅的开发体验。Ruby 开发人员会在使用 Pry 时有类似的体验，但是我不确定是否大部分 Java 开发人员会欣赏它。当然，他们的 IDE 在某种方式上弥补了像 Clojure 或 Ruby 这样的动态语言难以实现的方面，所以我猜这是种权衡，不过拿什么我都不换 REPL。

7.5 总结

和本书中其他框架相比，Immutant 是很不同的一种框架。它比通常的网络框架提供了更多的功能，并集成了如分布式缓存、消息队列、计划任务和集群之类的企业级功能。它还有很多独特的功能，比如对多语言编程叠加和 Clojure 的动态集成的支持。

现代网络应用经常使用缓存和消息机制，但它们不得不采用第三方服务或将一些工具组合起来达到目的。Immutant 充分集成这些功能并保证它们无缝衔接工作在一起，以此彻底简化了这些功能。这让你的关注点能放在你的应用逻辑上，而不是放在将服务和应用层整合起来上。

7.5.1 Immutant 的强项

Immutant 是基于 JBoss 的久经沙场的企业级 Java 应用服务器。它去掉了繁冗的形式、XML 和工作在 Java EE 应用上令人头疼的地方，并使得 JBoss 的高级功能得以完全保留及可以简单方便地被使用。

集群、分布式缓存和消息队列是很多现代应用的关键性功能，并且当你的应用更加流行时，会需要处理更多的数据和服务更多的用户，它们会对你的应用产生很大的帮助。将这些功能构建入框架可以让你把注意力放在更想关注的地方。

通过对 nREPL 客户端内置的支持，交互式开发成为了一件轻松的事情。所有的 Immutant 功能都可以通过 REPL 访问并且保持着 Clojure 开发的活力。通过和 Immutant REPL 进行一些会话后，你很难想象这场景背后就是一个 Java 企业级应用服务器。

7.5.2 Immutant 的弱项

Immutant 是在一个很多人不太熟悉的函数式语言 Clojure 上构建的。Immutant 的数据结构和功能组合是强有力的工具，但需要一些时间来熟悉。

因为它关注在高级服务上，Immutant 确实没有很多内置功能用来处理网络请求，除了支持 Ring 处理器。你需要选一些有用的 Clojure 库来完成这些工作。

7.5.3 最后的思考

通过 Immutable 构建应用感觉像是在作弊。不但对大公司使用的高级功能唾手可得，它还有像很多脚本语言一样的动态环境，并且有 Clojure 的所有功能。只用几行代码，你就可以构建出大规模应用而不会经常遇到麻烦和整合代码。Immutable 让曾经值得夸耀的事情完成得如此轻松，甚至你还没有意识到的时候就已经完成了。

第 8 章

结束

我们当然可以来讨论数百种有趣的 Web 框架，但是最终还是得在某个时刻停下来。现在是我们编写 Web 应用的创意空间之旅结束的时候了，我希望你已经学到了很多，受到了很大的启发，并且准备将这些有意思的东西应用到实际的项目中。

和其他很多事情一样，编程是一个不断妥协的过程，可能是内存与性能间的较量，也可能是类型安全和原型速度间的妥协。没有完美的 Web 框架，但是这些工具中妥协的思想非常有用，你可以从中学到更多。没有哪两个 Web 框架是一样的，也正因为如此，Web 开发人员才可以从一大堆的框架中选择出对自己项目最有用的那个。

在你开始自己的征途之前，我们来回顾一下这本书中讲到的一些关键点。

8.1 关键想法

贯穿本书，我们看到了很多种创建应用程序的方式，但是这些方式都有一些基本的共性。

- 简单性。
- 代码运行在何处——客户端、服务器端，或者一个大的集群上。
- 组合——从小的片段构建大的应用。

- 声明胜过指令——描述要做什么，而不是怎么做。
- 不同的类型系统。
- 状态机。
- 互动性。

我们来简要地重述一下这些关键点，以及体现这些点的对应的框架。

8.1.1 简单性

简单性是一个很重要的特性，因为人类对于理解软件复杂性的能力非常有限。简单性一般可以通过最小化或转换一个问题，或者关注一个问题的某些局部来获得。在本书中，这两种策略都有所体现。

Sinatra 采用极简主义来保证一切都尽可能简单。它使用了 Ruby 的富有表达力的语法来创建优美的 DSL，并据此来表述 Web 应用；它关注在一个很小的功能集合，这些功能都是非常常用的。它本身并不提供太多的特性，也正是由于它的简单，使得它可以很容易与其他工具整合。

Ring 把 HTTP 请求和响应直接转化为简单的数据结构。这种转化使得 Web 可以被 Clojure 中丰富的工具集来操纵和抽象。与图形编程技术中将模型转化为不同的坐标系不同，复杂的操作只是一系列简单、定义好的转化的组合。

与这些框架不同的是，Webmachine 尝试拥抱 HTTP 的复杂性，并展示了自己的权力与荣耀。

8.1.2 代码运行在何处

目前的 Web 应用对于运行在何处上有很大的灵活性。浏览器已经足够强大，大部分工作可以运行在客户端。Web 应用已经足够流行，一台独立的服务器已经不再够用。我们学习了可以运行在客户端的框架，也学习了可以伸缩到多个服务器上的后端框架。

CanJS 和 AngularJS 是可以完全运行在浏览器上的前端框架。HTML 模板, URL 路由以及所有的逻辑都发生在客户端, 数据持久则通过后端服务器的 API 通信来完成。这种分离是一种分离关注点的高效方式, 前后端可以通过公用一个接口来迅速迭代。

借助于 Haskell 强大的本地代码编译器, Yesod 最大程度地优化了执行速度, 不过它很大程度上还是一个运行在单机上的传统框架。

Immutable 可以将你的应用运行在一个集群上, 可以通过添加机器来扩展可用的资源。运行在多个系统上, 可以帮助提高系统在有机故障时的抵抗力, 缓存数据也被冗余到集群中的多台机器上。所以只要有一个机器还在运行, 整个系统就会一直工作。

8.1.3 组合

将小的块组合起来完成一个解决方案是软件开发中的一种常用模式。函数式编程语言诸如 Clojure 和 Haskell 非常强调这一点, 并且提供了很多综合的支持。Ring 和 Immutable 都提供了有趣的方式来实现组合。

在 Ring 中, 对于请求或者响应中数据的简单转换可以被组合在一起来形成一个管道。这样可以方便将中间件混合起来并组装成你所需要的管道。请求/响应映射同样是对数据不同程度的组合, 这样 Ring 和它的中间件就可以使用这些组合来完成自己的工作。

Immutable 中的管道是将小的功能组合起来形成一个大的整体的绝佳范例。管道中的每一步都有一个小的任务, 多个步骤可以被组织在一起来构建出一个可扩展的工作流, 并且可以在多核或者多机器上处理数据。

8.1.4 声明式优先于指令式

在很多框架中, URL 的路由都是声明式定义的。你现在只需要写一个关联了匹配模式和控制器的表, 而不是通过写一大堆的条件语句来判断路径, 然后分发到对应的处理函数。框架自身会根据这个表中的规则来自动转发。

这种描述了做什么而不是怎么做的方式在很多问题上都是实用的。在 Immutable

和 Ring 框架的组合中描述了这种方式，但是更详细的讨论在本书的 AngularJS 一章中。

AngularJS 对 HTML 进行了扩展，加入了声明式的语法来嵌入信息，它允许开发人员对词汇表进行扩展。这种做法让你关注在要做的事情上，然后框架来处理如何做。即使在开发人员既要关注做什么，又要关注如何做的情况下，这种方法仍然是有益的：这些不同的片段可以被独立地处理。

8.1.5 动态类型和静态类型

类型系统是业界最为古老的争论之一，也是最为人们所误解的。Web 框架一般都构建在动态类型的语言如 Ruby、Python 和 Clojure 之上。当然有很多框架是建立在静态语言如 Java 之上的。我们学习了 Yesod，它是用 Haskell 编写的，并且有着最为有趣且强大的类型系统。

动态语言中可以轻松地构建多种多样的数据类型。Ring 利用了这一点，它将 HTTP 请求转换为一个简单的映射表。静态类型的语言也可以做类似的转换，但是它们不得不创建一些特定的数据类型。

Yesod 使用了 Haskell 的类型系统来实现安全性约束。比如那些基于字符串注入的攻击天生就被防止了。用户输入的字符串和数据库看到的字符串根本就是不同的类型，也很难被混淆。正如我们在 Yesod 那一章看到的，类型可以被用来防止数据库查询中错误的 ID，还可以被用来编码业务逻辑的需求。

8.1.6 状态机

状态机是计算机科学中的一个抽象，在很多情况下都很有用。处理 HTTP 请求可以被很好地建模为一个状态机。Webmachine 的创造者有这样的洞察力，并且揭示了状态机应该被实现为回调函数。这样使开发者可以很容易地利用 HTTP 协议的强大特性，又不至于深陷入其复杂性。

Webmachine 应用程序只回答简单的问题：即决定状态机中路径。通常来说，这些回答就是简单的是或者否；当然也可以很复杂，比如返回所有被支持的内容类型的

列表。因为 Webmachine 处理了 HTTP 的复杂性，而只暴露简单的决策，开发人员就无须记住状态码或者其他 HTTP 协议里的奥秘，仅仅专注于领域功能和向 Webmachine 提问题即可。

8.1.7 交互性

开发操作系统原生的应用程序时，基本的开发流程是编辑、编译、运行，然后重复。Web 应用有类似的工作流程：编辑、刷新，然后重复。像 Lisp 这样的动态语言缩短了这个步骤，它们允许开发人员工作在一个 REPL 的环境中，并且可以在系统运行的同时进行编码。Yesod 和 Immutant 都尝试了将这种交互性代入 Web 开发中。

Yesod 虽然是一个原生的编译语言，但是可以监视你的修改，并且实时地重编译。这给我们熟知的编辑-刷新模式添加了像静态类型语言那样的特性。

Immutant 拥有 Lisp 的血统，它直接在你的 Web 应用中植入了一个 nREPL 服务器。你可以直接连接到这个服务上，修改并查看运行状态，甚至可以在运行期动态添加新功能。这种强大的功能可以彻底改变你的工作习惯。

客户端的交互性则又是另一回事儿了。传统上大部分 Web 框架是运行在服务器端的，客户端通过提交表单或者使用 Ajax 请求来和服务器进行交互。CanJS 和 AngularJs 则将应用程序挪到了客户端，为用户提供了无与伦比的动态响应。结果就是，这种方式开发的应用程序有着本地应用的响应速度，并且又包含了所有 Web 应用程序的优势。

8.2 快乐的探索吧

我们希望你在这次探险中已经学到了足够多的知识，也希望已经激发了你学习 Web 开发中的新想法。你可以找到很多拥有独特想法的、不同方向的框架，并由此找出开发应用的更好的方式。

我们的行业是一个每天都会高速变化的行业。过去的 15 年里，我们使用 Web 技术经历了静态页面（可能还包含一些 CGI 脚本）到文本处理器，到高质量的视频游戏，再到整个操作系统。而且这种 Web “吞食世界” 的趋势并没有任何停止的迹象。

作为开发人员，我们需要不断地扩展边界，令最终用户满意。探索开发技术的尖端是为你未来做准备。

继续搜索完美的框架吧，你可能永远都找不到它，但是在这个无畏的探索中你会得到非常多的收获。

七周七Web开发框架

Seven Web Frameworks
in Seven Weeks
Adventures in Better Web Apps

Web应用程序的迅速发展迫切需要创新的解决方案。了解各种框架以及它们独一无二的特性，将会启发并促使你在面对日常工作中的挑战时从一个新的角度去思考。

本书涵盖了7种影响现代Web应用并改变了Web开发方式的框架：Sinatra、CanJS、AngularJS、Ring、Webmachine、Yesod和Immutable。每一种框架都为构建Web应用带来了独特而强大的思路：

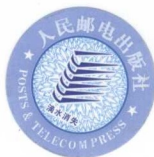
- 拥抱Sinatra的简单性，它摒弃了大框架的繁复，转而回归Ruby。
- 在客户端依赖CanJS，你可以在浏览器中使用JavaScript来创建完整的应用。
- AngularJS强调声明式编程，将声明式的HTML和JavaScript混合起来，只需要说你想要什么，而无需关心具体怎么做。
- Ring将Web变成数据，再使用Clojure来轻松操纵这些数据。
- Webmachine会让你成为高级HTTP的行家里手，并专注于Erlang的强大特性。
- 使用Yesod来证明Haskell的高级类型系统不只是学院派的玩具。
- 使用高端的Immutable绝对是企业级框架的明智之举。

不论你现在使用的是哪种框架，这7种框架都会影响到你的工作。

作者简介

Jack Moffitt使用不同语言、不同框架开发Web相关应用长达十年。他是Mozilla研究所的高级研究工程师，他工作于Servo项目，正在开发一个试验性的新一代浏览器引擎。他还帮助创建了Ogg Vorbis格式，并且创建了Xiph.org基金会，这是一个非盈利的机构，致力于开源且完全免费的多媒体编解码器。

Fred Daoud是一位非常有热情的Web程序员，他喜欢尝试不同的新框架。他还是 Stripes ...and Java Web Development Is Fun Again和Getting Started With Apache Click这两本Web框架书籍的作者。作为Modernizing Medicine的软件工程师，他使用Stripes、jQuery、YUI和CanJS进行日常开发。



ISBN 978-7-115-38843-8

定价：59.00 元

分类建议：计算机 / 程序设计 / Web开发
人民邮电出版社网址：www.ptpress.com.cn